Lotus

SmartSuite 97 EDITION

The Best Suite For Today's Connected World — Powered by 1-2-3.

DEVELOPING SMARTSUITE APPLICATIONS USING LOTUSSCRIPT

WINDOWS 95 and NT 4.0

Some of the products referenced appear in online documentation.

The company names appearing in this book are fictitious.

Contents

Chapter 4 Building a Single-Product Application 4-1

Chapter 5 Building Cross-Product Applications 5-1

Chapter 6 Integration with Notes 6-1

Chapter 9 Using LotusScript in Freelance Graphics 9-1

Chapter 10 Using LotusScript in Word Pro 10-1

Introduction

Developing SmartSuite Applications Using LotusScript describes how to use LotusScript® to develop applications in the following Lotus® products:

- Lotus 1-2-3® 97 Edition for Windows® 95 and Windows NT™ 4.0
- Lotus Approach® 97 Edition for Windows 95 and Windows NT 4.0
- Lotus Freelance Graphics® 97 Edition for Windows 95 and Windows NT 4.0
- Lotus Word Pro® 97 Edition for Windows 95 and Windows NT 4.0
- Lotus Notes®

Who should read this book

Developing SmartSuite Applications Using LotusScript is for both new and experienced application developers. It contains basic information about the application programming interfaces (APIs) of 1-2-3, Approach, Freelance Graphics, and Word Pro, and introduces new developers to the concepts of programmable objects and how to manipulate them. It also includes information on how to use the Integrated Development Environment (IDE) and Dialog Editor that Lotus provides.

If you're already familiar with developing applications using products such as IBM® VisualAge™ or Microsoft® Visual Basic®, you will be comfortable with the way LotusScript programming tools support the development process. If you are experienced in developing applications in Lotus SmartSuite® 97 Edition for Windows 95 and Windows NT 4.0 products using macro commands, *Developing SmartSuite Applications Using LotusScript* will help you switch from a macro-based development environment to a structured programming language environment that enables object-oriented application development within and across products.

Before reading *Developing SmartSuite Applications Using LotusScript*, you should be familiar with basic Windows 95 concepts and techniques. For more information, see your Windows documentation.

Using this book with Help

Developing SmartSuite Applications Using LotusScript and online Help complement each other. Help documents the LotusScript language and every LotusScript language extension in 1-2-3, Approach, Freelance Graphics, and Word Pro. It also provides extensive IDE and Dialog Editor Help.

To access Help, follow these steps:

Accessing the LotusScript Index from your product

1. Choose Help - Help Topics.
2. Search on “LotusScript” in your product Help index.

   The Help topic “Overview: Using LotusScript” is displayed.
3. Click “LotusScript Index.”

Accessing the LotusScript table of contents from your product

1. Choose Help - Help Topics.
2. Click the Contents tab.

   The Help table of contents is displayed.
3. Click “LotusScript.”

Accessing Help from the IDE

1. Choose Help.
2. Do one of the following:
   - Choose LotusScript or *product* Objects and search on the specific term you want Help on.
   - Choose Script Editor.

Accessing Help from the IDE Browser

1. Choose the class or class member name you want Help on.
2. Press **F1**.

Accessing Help from the Dialog Editor

1. Choose Help.
2. Choose Dialog Editor or Lotus Controls.

Additional LotusScript documentation

In addition to this manual and online Help, you can refer to the following materials for information on LotusScript. All of the following, with the exception of the LotusScript Extension (LSX) Toolkit, are available in hardcopy. They are all available in Adobe Acrobat® or HTML formats in your SmartSuite 97 package, in the SmartSuite Application Developer's Documentation Set, or on the LotusScript home page on the World Wide Web (http://www.lotus.com/smartsuite/sslotusscript.htm).

- *Getting the Most Out of LotusScript in SmartSuite 97,* a brief overview that explains how SmartSuite 97 products use the LotusScript programming language and how your business can take advantage of LotusScript in developing applications for SmartSuite.
- The *LotusScript Programmer's Guide*, a general introduction to LotusScript that describes its basic building blocks and how to put them together to create applications.
- The *LotusScript Language Reference*, a comprehensive summary of the LotusScript language, presented in A-Z format. The *LotusScript Language Reference* is also available as Help in all Lotus products that support LotusScript.
- The LotusScript Extension (LSX) Toolkit, a source of information about developing LotusScript Extension (LSX) modules. LSX modules are Dynamic-Link Libraries (DLLs) that contain public class definitions for any product that supports the LotusScript language. Typically LSX modules add classes to your SmartSuite product, effectively expanding the number of classes available for all your scripts. The Toolkit is designed for developers with some experience in the C or C++ programming language.

Organization

Developing SmartSuite Applications Using LotusScript has 10 chapters.

- Chapter 1, "SmartSuite Applications: An Overview," contains basic information about the SmartSuite development environment.
- Chapter 2, "LotusObjects: Building Blocks for Developing Applications," explains in detail what LotusObjects are and how to manipulate them using their properties, methods, and events. It describes the object-oriented programming concepts of containment and inheritance and what they mean for the applications you develop.

- Chapter 3, “LotusScript Programming Tools,” introduces the IDE and Dialog Editor and describes how you can use these tools throughout the development process to design applications, write code, run and debug applications, and develop reusable code libraries for multiple applications.
- Chapter 4, “Building a Single-Product Application,” builds on the information in Chapters 2 and 3 to explain how to develop an application. The chapter presents four different versions of a sample application that runs in Word Pro to illustrate the various ways you can design and run an application.
- Chapter 5, “Building Cross-Product Applications,” explains how to use OLE Automation to develop an application that requires two or more SmartSuite products. The sample application runs in Word Pro and uses Approach data. This chapter also includes a sample application developed in Visual Basic that manipulates objects in SmartSuite.
- Chapter 6, “Integration with Notes,” describes how to use OLE Automation and the Notes LSX to control SmartSuite products from scripts running in Notes™, and Notes from scripts running in SmartSuite products.
- Chapter 7, “Using LotusScript in 1-2-3,” describes the 1-2-3 object model and explains how to develop applications in 1-2-3. The chapter contains a collection of sample applications that illustrate application development concepts.
- Chapter 8, “Using LotusScript in Approach,” describes the Approach object model and explains how to develop applications in Approach. The chapter contains a collection of sample applications that illustrate application development concepts.
- Chapter 9, “Using LotusScript in Freelance Graphics,” describes the Freelance Graphics object model and explains how to develop applications in Freelance Graphics. The chapter contains a collection of sample applications that illustrate application development concepts.
- Chapter 10, “Using LotusScript in Word Pro,” describes the Word Pro object model and explains how to develop applications in Word Pro. The chapter contains a collection of sample applications that illustrate application development concepts.

Conventions used in this book

Developing SmartSuite Applications Using LotusScript uses the following conventions.

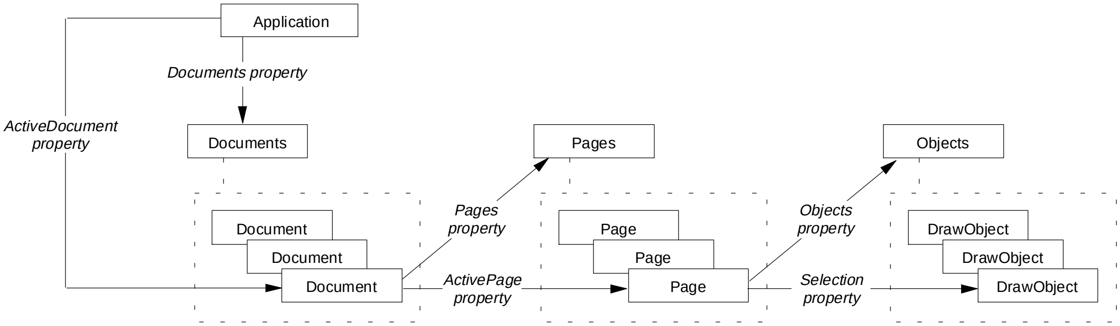
Object model diagrams

Chapters 2, 7, 8, 9, and 10 present object model diagrams to illustrate partial views of the containment and inheritance hierarchies in each of the products. The LotusObjects™ appearing in these diagrams are those that you are most likely to use in the applications you develop.

The following diagram conventions are used to illustrate object relationships.

Containment diagrams

A containment relationship is shown by joining two boxes with an arrow. For example, in the following diagram, the Application class, represented by the box labeled Application, contains the Documents class, represented by the box labeled Documents. Application contains Documents through the Documents property, as indicated by the label Documents property that appears on the arrow connecting Application and Documents.

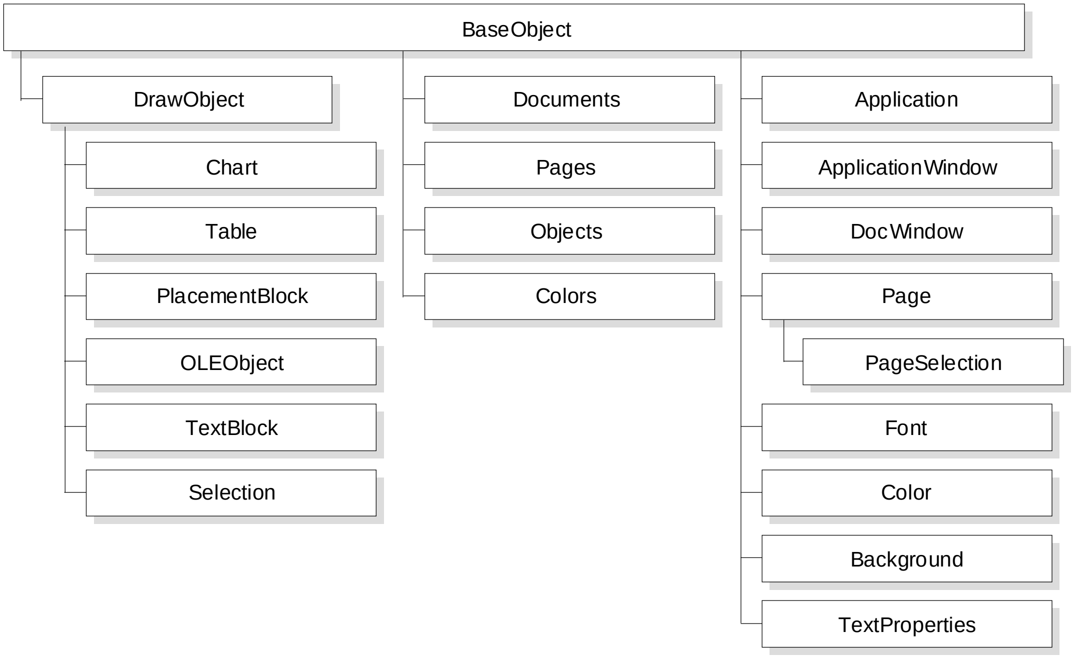


Documents is a collection class, as indicated by the dotted line leading to three boxes labeled Document, all of which are enclosed in a dotted-line box. This illustrates that Documents is made up of a collection of individual Document objects.

The Application class also contains the Document class, as indicated by the arrow pointing to an individual box labeled Document. Application contains Document through the ActiveDocument property, as indicated by the label, ActiveDocument property, appearing on the arrow connecting Application and Document.

Inheritance diagrams

An inheritance relationship is shown by using a line to connect a class to one or more of the classes that inherit members from it. For example, the following diagram illustrates that the following objects inherit from BaseObject: DrawObject, Documents, Pages, Objects, Colors, Application, ApplicationWindow, DocWindow, Page, Font, Color, Background, and TextProperties. BaseObject is the base class for all of these classes.



This diagram also illustrates that Chart, Table, PlacementBlock, OLEObject, TextBlock, and Selection all inherit from DrawObject. Also, PageSelection inherits from Page.

Sample files

Developing SmartSuite Applications Using LotusScript uses examples to illustrate application development concepts for each of the products. These examples, described in the text, are available in online files so that you can run them and view the code in the IDE. When SmartSuite is installed, the files are copied to the default sample files directory C:\LOTUS\SAMPLES\SUITE. However, because the default directory for SmartSuite can be changed at Install time, throughout this book files are referred to only by the file name and a reference to the "sample files directory," without any reference to a specific path.

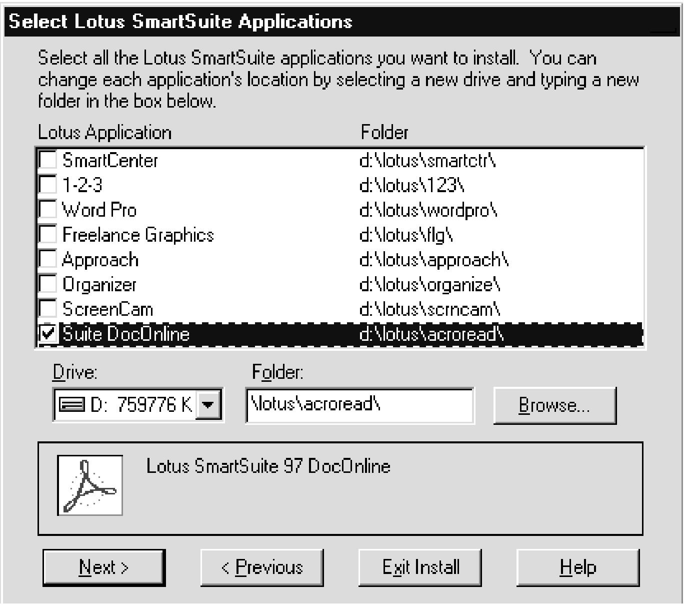
For example, DW04_S1.LSS refers to a file called DW04_S1.LSS stored in the sample files directory, which is either C:\LOTUS\SAMPLES\SUITE or another directory specified at Install time. Check with your system administrator to verify the name of the path to which your copy of SmartSuite is installed.

Installing the sample files

When you install SmartSuite, the sample files are not automatically installed. You have to specify that you want them downloaded by performing a customized Install. You can rerun the Install program at any time to download the sample files.

To install sample files when you run Install:

1. In the Select Lotus SmartSuite Applications dialog box, select Suite DocOnline and then click Next.



Select Lotus SmartSuite Applications

Select all the Lotus SmartSuite applications you want to install. You can change each application's location by selecting a new drive and typing a new folder in the box below.

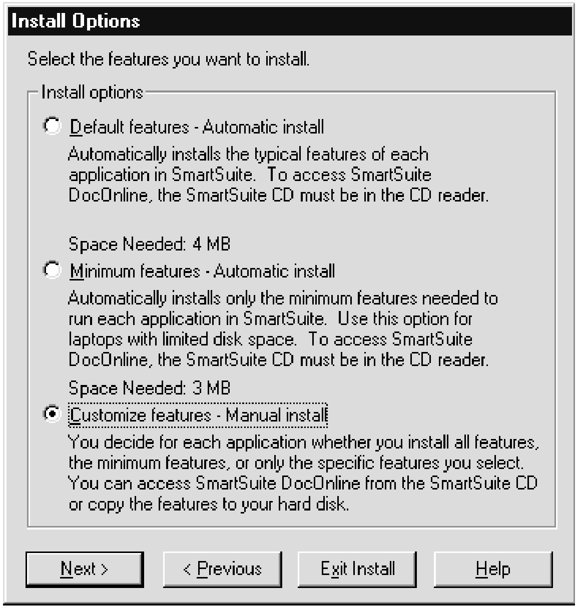
Lotus Application	Folder
□ SmartCenter	d:\lotus\smartctr\
□ 1-2-3	d:\lotus\123\
□ Word Pro	d:\lotus\wordpro\
□ Freelance Graphics	d:\lotus\flg\
□ Approach	d:\lotus\approach\
□ Organizer	d:\lotus\organize\
□ ScreenCam	d:\lotus\scrncam\
☑ Suite DocOnline	d:\lotus\acroread\

Drive: D: 759776 K

Folder: \lotus\acroread\

Lotus SmartSuite 97 DocOnline

2. In the Install Options dialog box, select Customize features - Manual install and then click Next.



Install Options

Select the features you want to install.

Install options

○ Default features - Automatic install

Automatically installs the typical features of each application in SmartSuite. To access SmartSuite DocOnline, the SmartSuite CD must be in the CD reader.

Space Needed: 4 MB

○ Minimum features - Automatic install

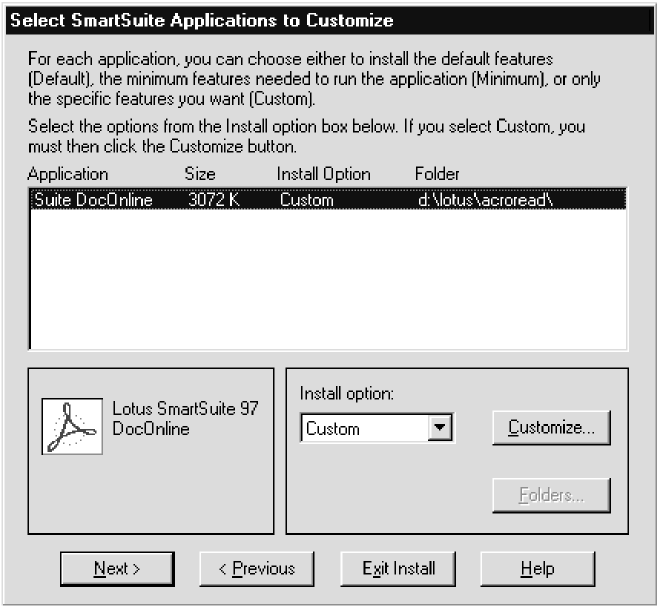
Automatically installs only the minimum features needed to run each application in SmartSuite. Use this option for laptops with limited disk space. To access SmartSuite DocOnline, the SmartSuite CD must be in the CD reader.

Space Needed: 3 MB

● Customize features - Manual install

You decide for each application whether you install all features, the minimum features, or only the specific features you select. You can access SmartSuite DocOnline from the SmartSuite CD or copy the features to your hard disk.

3. In the Select SmartSuite Applications to Customize dialog box, click Customize and then click Next.



Select SmartSuite Applications to Customize

For each application, you can choose either to install the default features (Default), the minimum features needed to run the application (Minimum), or only the specific features you want (Custom).

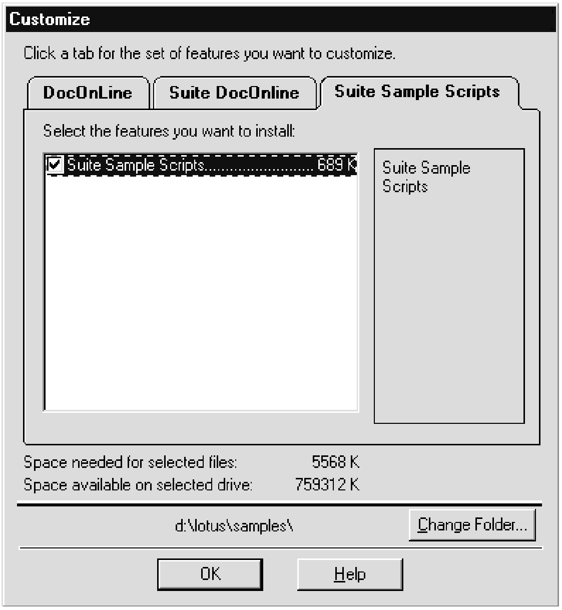
Select the options from the Install option box below. If you select Custom, you must then click the Customize button.

Application	Size	Install Option	Folder
Suite DocOnline	3072 K	Custom	d:\lotus\acroread\

Lotus SmartSuite 97 DocOnline

Install option: Custom

4. In the Customize dialog box, click the Suite Sample Scripts tab and select Suite Sample Scripts.



Click a tab for the set of features you want to customize.

Select the features you want to install:

☑ Suite Sample Scripts......................... 689 K

Suite Sample Scripts

Space needed for selected files: 5568 K

Space available on selected drive: 759312 K

d:\lotus\samples\

5. Click OK to install the sample files.

Chapter 1
SmartSuite Applications: An Overview

SmartSuite offers a sophisticated development environment to help you build applications using a variety of Lotus products. With LotusScript, a powerful BASIC programming language, you can customize, automate, and integrate the following products to create new and robust applications of your own:

- 1-2-3®
- Approach®
- Freelance Graphics®
- Word Pro®
- Notes™

The power and flexibility of all these Lotus products allow you to meet your specific business needs and make you and your users more productive.

Development needs that are changing

The current environment in business application development demands greater ability to connect different workgroups and products and to create more powerful and highly sophisticated applications while paying close attention to a greater return on investment. This is influenced and supported by the following technologies:

- Desktop suites
- Object-oriented technology
- Team computing

SmartSuite as an application development platform

Stand-alone products like 1-2-3, Approach, Freelance Graphics, and Word Pro offer valuable features for specific business needs. Most users working in these products take advantage of these features to perform repeated tasks. Your job, as a developer of applications, is to automate these tasks in the products in which they are performed, thus making your users more productive.

These products can also be powerful for users when treated as a single platform upon which an organization standardizes its processes. For you, the developer, this translates to a collection of objects that you can recombine and program across product boundaries to build task-specific functions and new applications.

The SmartSuite desktop products can be integrated easily not only because they have compatible user interfaces and functionality, but because they share a common programming language, similar application programming interfaces (APIs), a common editor and debugger for writing and maintaining code, and a common dialog editor for designing custom user interfaces.

LotusScript: a common language

SmartSuite uses LotusScript, a version of BASIC that offers not only the standard capabilities of structured programming languages, but a powerful set of language extensions that enable object-oriented development within and across products as well. Using LotusScript, you can combine features of Notes and the various SmartSuite products to customize, automate, and integrate these products as well as to create new applications.

LotusObjects: the SmartSuite API

1-2-3, Approach, Freelance Graphics, and Word Pro share similar APIs, composed of LotusObjects. LotusObjects represent millions of lines of professionally written and tested code that you can combine into your own applications. These objects share a common design. For example, the Application object is virtually the same in all the SmartSuite products. This similarity allows for easy access to objects from one product to another. Other objects are unique to the individual APIs but can be used in cross-product scripts because their design is based on a common standard recognized by all of the Lotus products.

The existence of common objects provides an efficient arena for developing and implementing applications. LotusObjects expose the internal functions of these products so that you can develop applications. Furthermore, you can design your own objects, either based on or independent of LotusObjects, using LotusScript. All LotusObjects are OLE Automation objects, allowing for greater ease in cross-product scripting.

Shared tools

SmartSuite products share a common set of programming and design tools: the Integrated Development Environment (IDE), for creating, editing, and debugging scripts, and the Dialog Editor, for designing custom dialog boxes.

Using the IDE, available in all SmartSuite products from the Edit pull-down menu, you can create, edit, and debug scripts for any of the Lotus products installed on your system. You can store, compile, and execute scripts that are associated with specific LotusObjects and run them from inside SmartSuite or Notes.

The Dialog Editor enables you to design custom dialog boxes containing all the traditional dialog box controls, such as command buttons, list boxes, option buttons, and check boxes. These controls are implemented as OLE controls (OCXs). You are not limited to using the controls supplied with the Dialog Editor—you may use any standard OCX.

Using LotusScript, LotusObjects, the IDE, and the Dialog Editor, you can develop rich graphical user interfaces that allow your application to interact with your users.

For more information about the IDE and the Dialog Editor, see Chapter 3.

Object-oriented technology

More and more companies are providing application development tools and products based on object-oriented technology, and SmartSuite is no exception. The use of objects provides great opportunities for code portability, reuse, and customization. Application developers can write new applications that are easy to share, maintain, and develop as a team, using objects from one or more products.

OLE

Object Linking and Embedding is a technology that takes advantage of object-oriented application design by letting you use LotusObjects and objects from non-Lotus products to combine and exchange information across product lines.

OLE Automation

OLE Automation is an OLE technology that allows objects to be manipulated from outside the application in which they reside. All LotusObjects are OLE Automation objects, which means that they can be exposed to different Lotus and non-Lotus products. LotusScript in SmartSuite uses OLE Automation to expose LotusObjects to Notes and SmartSuite to other applications. In turn, OLE Automation controllers such as Visual Basic, can access LotusObjects.

For more information about using OLE Automation with LotusObjects and user-defined objects, see Chapter 5 and Chapter 6.

SmartSuite and Notes integration

Notes is a group information manager that enables teams to be more effective in collecting, organizing, and sharing information across local and wide-area networks and dial-up lines. Notes database applications help diverse groups communicate and work as a team.

SmartSuite offers the best integration with Notes available in today's marketplace for the following reasons:

- The two products have compatible APIs.
- SmartSuite products can access the Notes LSX, which exposes Notes objects, to use Notes for backend database functionality. This enables you to use the SmartSuite products as an alternate user interface for Notes.
- SmartSuite and Notes share the same programming language (LotusScript) and similar development tools.
- Notes/FX™ enables SmartSuite products and Notes to share information at the field level, using Notes/FX fields.

For more information about Notes and SmartSuite integration, see Chapter 6.

Team Computing

Today the success of businesses depends heavily on the ideas and contributions of all team members. The Lotus Team Computing features help teams improve productivity by making it easy to gather and distribute information. You can write scripts that automate many of the Team Computing features using the product APIs and then include the scripts in the SmartSuite applications you develop.

Using the Lotus Team Computing technologies, SmartSuite is the first team desktop suite on the market. SmartSuite enables users to better collaborate on and share documents, spreadsheets, databases, presentations, and more, by improving the way people can create, review, and edit all types of business information. Team Computing reduces the time to complete business tasks and reduces the cost of collaboration.

Team Computing features include TeamReview™ in Word Pro and Freelance Graphics, TeamShow™ in Freelance Graphics, TeamConsolidate™ in Word Pro, Versioning and TeamMail™ in 1-2-3, and TeamSecurity in all the SmartSuite products.

Some of these features are Notes-enabled, adding to the tight integration between Notes and SmartSuite. Furthermore, through the TeamMail feature, you can easily access any VIM-enabled or MAPI-enabled mail package from within all the SmartSuite products as well as from any SmartSuite application.

For more information about automating these Team Computing features in the products, see Chapters 7 and 10.

Chapter 2
LotusObjects: Building Blocks for Developing Applications

This chapter presents an overview of LotusObjects and how you use them to develop applications. The chapter provides a basic, conceptual introduction to object-oriented programming. The concepts presented here are used throughout the rest of the book. If you want to go directly to practical information about applying these concepts in each of the products, see Chapters 6 through 10 where specific code examples are presented and explained.

Objects

Objects are the building blocks of any application. An object represents a part of a product that you manipulate in a script, using its members, which are the methods, properties, and events that apply to it. For example, the 1-2-3 Range object represents a range in a sheet. You might want to write a script that checks the data in a range and performs actions based on the values in that range. To do this, you use the Range object along with its methods, properties, and events.

Consider objects as the building blocks used to develop an application and the LotusScript language as the hands that manipulate and put the blocks together. Using the LotusScript language, you can program SmartSuite objects, called LotusObjects, to control your applications in creative and sophisticated ways.

All LotusObjects share a common design. Many objects are implemented either the same way across products, or almost the same way, with slight variations from product to product. For example, the Application object is implemented the same way in both Approach and Word Pro. However, in Word Pro the Color object has certain characteristics that the Approach Color object does not. The advantage of a common design is that it lends itself to ease of learning and ease of use. The time you invest and what you gain in learning about the LotusObjects in one SmartSuite product can be carried over into another product.

Objects and classes

An object is an instance of a class. For example, an Approach FieldBox object is a specific instance of the Approach FieldBox class. It represents an existing field box on one of your Approach forms. If there are four field boxes on your Approach form, then there are four FieldBox objects, but only one FieldBox class.

A class is a description or a definition of a part of a SmartSuite product. It has members, which are its properties, methods, and events. These are the characteristics and behaviors of the objects instantiated from the class. For example, Top (a property), Refresh (a method), and Click (an event) are members of the Approach FieldBox class. They are characteristics of all FieldBox objects. A class is like a specification of an object.

Classes are identified by their names, for example, the 1-2-3 Rectangle class. Objects, however, are generally referred to in scripts by the name of the variable, property, or parameter in which they are stored.

The easiest way to distinguish between classes and objects is to think of the class as the description of a part of a SmartSuite product, and the object as one instance of the part described by the class. That instance can be identified with its own unique name, which distinguishes it from all other instances of the class. To understand this concept, think of the word “movie,” which describes a class of entertainment. “Casablanca” is one instance of the class movie. “Gone with the Wind” is another instance of that class. “Casablanca” and “Gone with the Wind” are like objects because they are specific, real movies; whereas movie is like a class because it describes the characteristics of a movie without specifically naming any particular movie.

Collection classes

You will notice among the various classes available in SmartSuite products that the names of some classes end with the word “Collection” or are in the plural form. For example, Word Pro has a Bookmark class, but it also has a BookmarkCollection class. It has both a Document class and a Documents class. BookmarkCollection and Documents are collection classes. A collection class is what the name implies: a class made up of a collection of the objects of that particular class. Another way to think of a collection is as a wrapper around a group of objects of a class that together, as a group, form a separate object.

Collections are useful for writing programs because they group objects of the same class together and then allow you to step through, or iterate through, the collection to do things to each object in it, the way you would with an array or a list (arrays and lists are actually types of collections). You use the LotusScript ForAll statement to iterate through a collection. You can also index an element of a collection using the For statement.

For information about arrays and lists, see Chapter 3 of the *LotusScript Programmer's Guide*. For information about collection classes, see Chapter 9 of the *LotusScript Programmer's Guide*. For information about the LotusScript For and ForAll statements, search on "LotusScript" in your product Help index, click "LotusScript Index," then search on "For" and "ForAll."

The product object models

Every SmartSuite product also has its own set of unique LotusObjects that are not part of any other product's object set. Like the common LotusObjects, these product-specific LotusObjects represent programmable elements of a particular product. For example, FootNoteLayout is a Word Pro object that represents the layout of a footnote on a page in a Word Pro document. This object represents something in Word Pro, but has no meaning in any of the other SmartSuite products.

The way that a product object set, or object model, is organized also distinguishes it from the other product object models. The relationships between the objects vary from product to product. Each SmartSuite object model has its own containment and inheritance structure. For information about containment and inheritance, see the sections "Containment" and "Inheritance" later in this chapter.

SmartSuite user interface (UI) gestures for manipulating data have equivalent object behaviors, represented by properties, methods, and events. The power of programming in SmartSuite products lies in understanding the product objects and their behaviors so that you can achieve a desired effect in your application. In order to understand the objects and how they relate to each other, you need to understand each of the product object models. For diagrams and descriptions of the product object models, see Chapters 7 through 10.

Methods, properties, and events

Objects have members: the methods, properties, and events of the class that an object is instantiated from. The members allow you to manipulate objects in a variety of ways.

Methods

Methods are the actions performed by an object. These actions are the object's behaviors. For example, the Freelance Graphics Document object has a CreatePage method, which creates a new page in a document. It also has a Close method, which closes a document. A method is a LotusScript function (which returns a value) or a sub (which does not return a value).

Methods let you control your application by performing specific object actions. Whenever you want to perform an action, you invoke a method.

Properties

Objects have characteristics, called properties, that allow you to examine and control the state of an object, that is, how it looks and behaves. For example, the Word Pro Document object, which represents a document, has a ReadOnly property, which indicates whether or not the document is a read-only document. A read-only document behaves differently than a read-write document. The Word Pro Document object also has a FullName property, which defines the document name. You can use a document name in a script to manipulate the document. The Freelance Document object, which represents a Freelance presentation, has a ViewMode property, which indicates the mode, such as Outline, in which you can view a presentation.

A property is like a variable in that it stores a value. Thus, like a variable, a property has a data type, so the kind of value you can store in it depends on its data type. For example, the Word Pro object FullName property has a data type of String, so any value stored in it must be a text string.

You can specify, or set, the state of an object by setting the value of its properties. For example, the Freelance Document object ViewMode property has four acceptable values: $ViewDraw (Page view mode), $OutlineView (Outliner view mode), $ViewSorter (Page Sorter view mode), and $ViewSlideShow (Screen Show view mode). If you want to change the view mode of your presentation, you set its ViewMode property to the value that represents the view mode you want.

Some properties are read-only; you cannot change these properties by setting them directly in a statement. Instead, they change by a different means. For example, the Approach Document object, which represents an Approach .APR file, has a Modified property that is read-only. The value of the property, True or False, indicates if the .APR file has been modified. Since Modified is a read-only property, you cannot set its value directly to either True or False. It changes, however, when the user makes a change to the .APR file, via the user interface, or when a script sets a property or invokes a method of the Document object representing the file and thereby changes it.

Although you can’t *set* read-only properties, you can *get*, or read, their values in a script and use those values to perform other actions. For example, you might write a script that gets the value of an Approach Document object Modified property, and, if the value equals True, closes the document it represents and opens another one.

Events

LotusObjects are equipped with a rich set of events. An event is an action performed by a user, an application, or the system. For example, saving a document is a kind of event. Sometimes a document is saved by a user action (choosing File - Save) or by the application itself, via the autosave feature. Clicking a button in 1-2-3 is another kind of event that occurs when a user clicks a button on a sheet. Most objects have a set of predefined events with names that match the action of the event. For example, clicking a button is represented by the Click event of the Button object.

The Integrated Development Environment (IDE) provides a sub for an event. The name of the sub matches the name of the event. For example, the Click sub is where you write the event script for the Click event. When the user clicks a button, the Click event of the Button object occurs, and the code inside the Click sub is executed. Thus, you can control application processing and functionality by specifying what operations occur as a result of an event taking place.

Depending on the product, you can also initiate execution of a script from a macro, menu option, icon, or from another script; but the most common way is through an event, since events correspond to specific actions. For example, the Opened event of the 1-2-3 Document object takes place when you open a document, so it is easy and convenient to write a script that is associated with only that specific behavior. For information about viewing and using events in the IDE, see Chapter 3. For code examples that illustrate using events to run scripts in an application, see Chapter 4.

Dot notation: using methods and properties with objects

To manipulate an object in a script, you must indicate which method you want to invoke or which property you want to set or get. To do this, you use a syntax style called dot notation. Dot notation lets you easily refer to the members of a class.

In general, statements that use dot notation have the following structure:

`Object.Member`

Member is either a name of a property or of a method being invoked.

Dot notation and properties

To set a property, use a statement with the following structure:

`Object.Property = Value`

For example, the following statement uses dot notation to change the value of a property in 1-2-3. The statement sets the Name property of a Sheet object called Sheet1.

```
Sheet1.Name = "Expenses"
```

The text to the left of the dot, Sheet1, represents the Sheet object being manipulated. The text to the right of the dot, Name, represents the class member, in this case the Name property of the Sheet object. The text to the right of the equals sign, “Expenses,” is the value assigned to the Name property.

You can also get a value of a property using dot notation by assigning the value to a variable. For example, the following statement adds 1 to the value of a property (assuming the property has a numeric data type) and assigns the result to the variable CntVariable:

```
CntVariable = Object.Property + 1
```

Dot notation and methods

To invoke a method, follow the LotusScript rules for invoking functions and subs. For example, a statement with the following structure is a method that is a sub. You can use the same syntax for functions, but the return value is ignored.

```
Call Object.Method (<argument list>)
```

To use the return value of a method, invoke the method and assign its return value to a variable.

```
Variable = Object.Method (<argument list>)
```

For information about how to use functions and subs, see Chapter 4, *LotusScript Programmer’s Guide.*

The following simple example, from Freelance Graphics, uses dot notation to invoke methods that are subs:

```
Call CurrentDocument.Save
Call CurrentDocument.CloseWindow
```

The text to the left of the dot, CurrentDocument, represents the object being manipulated. In both statements, the text to the right of the dots names methods of that object, Save and CloseWindow, which are invoked in this script. The Save method saves any changes made to the object. The CloseWindow method closes the document window in which the document represented by CurrentDocument is displayed.

Leading dot notation

A statement that starts with a dot uses leading dot notation. The object to the left of the dot is omitted. For example, the following statement invokes the Save method of an object.

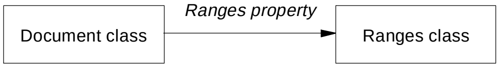
```
Call .Save()
```

All SmartSuite products resolve the leading dot to an object, but exactly which object is returned and which class it belongs to depends on the product. See Chapters 7 though 10 for information about how each product treats the leading dot.

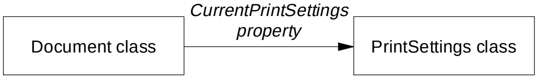
Containment

One way to view the SmartSuite product object models is through their containment hierarchies. A containment relationship exists between two classes when one class, the parent, has a property that contains an object of another class, the child. A property contains an object of a class when the property data type is that class.

For example, the 1-2-3 Document class has a property called Ranges. The value of this property is a Ranges object, which means that the Document class contains the Ranges class through the Ranges property. The Ranges property has a data type of the Ranges class.



The 1-2-3 Document class also has a property called CurrentPrintSettings. The value of this property is a PrintSettings object, which means that the Document class contains the PrintSettings class through the CurrentPrintSettings property. The CurrentPrintSettings property has a data type of the PrintSettings class.



As these examples show, a property name can be the same as the name of the contained class (Ranges property and Ranges class). However, this is not always the case (CurrentPrintSettings property and PrintSettings class).

These examples also show that a parent class can have several containment relationships (the Document class contains the Ranges class and the PrintSettings class). These relationships can be to objects of the same class or of different classes. For example, the 1-2-3 Background class has, among others, three properties, Color, BackColor, and Class, and therefore three containment relationships. It contains the Color class through the Color property, the Color class through the BackColor property, and the ClassInfo class through the Class property. The fact that Background contains the Color class twice is easy to understand if you think of the data types of properties. For example, Background also has two other properties, Description and Name, both with the data type of String. The BackColor and Color properties are merely two properties with the same data type, Color.

Similarly, a class can be contained by multiple classes. However, an object, a single instance of a particular class, can only have a single parent, and that parent is an instance of one of the parent classes. For example, in Approach, several different classes, including BodyPanel, SummaryPanel, and HeaderFooterPanel, contain the FieldBox class. This means that the FieldBox class has multiple parents, which are all classes directly above it in the containment hierarchy. (See the Approach object hierarchy diagram in Chapter 8 for details.) However, a FieldBox object can only have a single parent, and that parent is an instance of only one of the parent classes, for example, a particular SummaryPanel object.

For specific descriptions and diagrams of the containment hierarchy in each of the SmartSuite products, see Chapters 7 through 10.

Traversing the containment tree to access objects

Understanding which classes are contained by other classes, and therefore which objects are contained by other objects, helps you understand the syntax to use in a script when you try to access an object or change one of its properties. If you know a product containment hierarchy, you can access any object by traversing the containment tree that connects objects with other objects. To traverse the containment tree, use dot notation.

A statement that uses the structure of *Object.Member*, in which *Member* is a property that contains an object of a class, returns an object. For example, in all SmartSuite products, an Application object for the current application session is stored in the predefined global product variable CurrentApplication. The ActiveDocument property of an Application object stores a Document object. So, CurrentApplication.ActiveDocument, where ActiveDocument is the equivalent of *Member*, returns a Document object.

One of the Document object properties, ReadOnly, specifies whether the status of the document is read-only. To see if the active document is read-only, use an If... Then statement:

```
If CurrentApplication.ActiveDocument.ReadOnly Then
   ....
End If
```

In this example, CurrentApplication.ActiveDocument is evaluated and the appropriate Document object is returned. Then the value of the ReadOnly property of that object is checked.

There is no limit to the number of objects you can traverse in a statement such as this. If the Document object contains another object, and that object in turn contains an object that you want to access, you can add those names to the right of the dot after ActiveDocument, and separate each name in the containment hierarchy with a dot. The hierarchy tree is always evaluated from left to right as in the example above that starts with CurrentApplication and ends with ReadOnly.

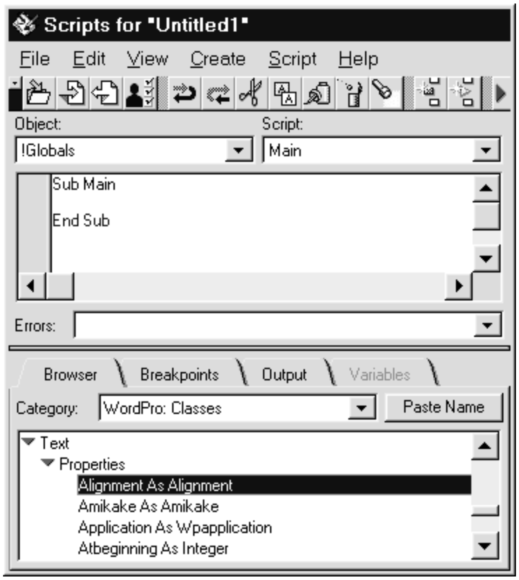
The following Word Pro example illustrates how to use the containment relationships between objects to set properties of an object. In the example, the script changes the alignment of a block of selected text.

```
CurrentApplication.Text.Alignment.AlignmentType = _
   $LtsAlignmentHorizCenter
```

To change the alignment, the statement uses the Alignment property of the Text object. Dot notation is used to separate the Text object from its Alignment property. Note that Alignment is also a Word Pro class, with its own properties, such as AlignmentType.

The IDE provides a Browser panel that helps you view containment relationships. (For information about the IDE Browser, see Chapter 3.) It lists all the classes and their members of the product in which you are viewing the panel. For example, if you look in the IDE Browser in Word Pro (illustrated next), you can view the Text class and all its members, including the Alignment property. Alignment is listed under text as “Alignment As

Alignment," which indicates that Alignment is a property of Text with a data type of the Alignment class. This means that the Text class contains the Alignment class through the Alignment property.



Object: !Globals

Script: Main

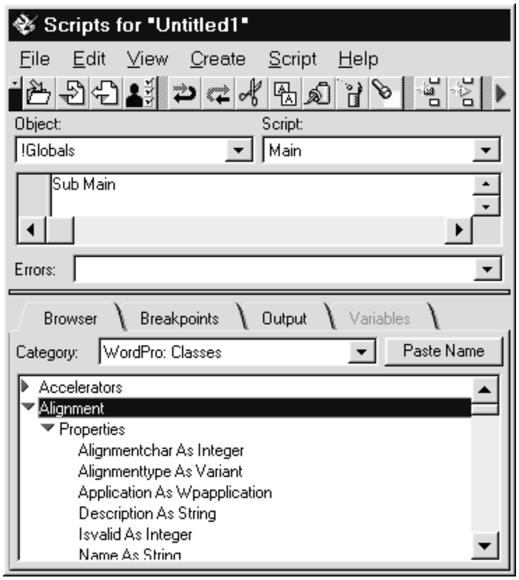
```
Sub Main

End Sub
```

Errors:

Category: WordPro: Classes

```
▼ Text
  ▼ Properties
      Alignment As Alignment
      Amikake As Amikake
      Application As Wpapplication
      Atbeginning As Integer
```

However, Alignment also appears as its own class with its own properties, including the AlignmentType property.



Object: !Globals

Script: Main

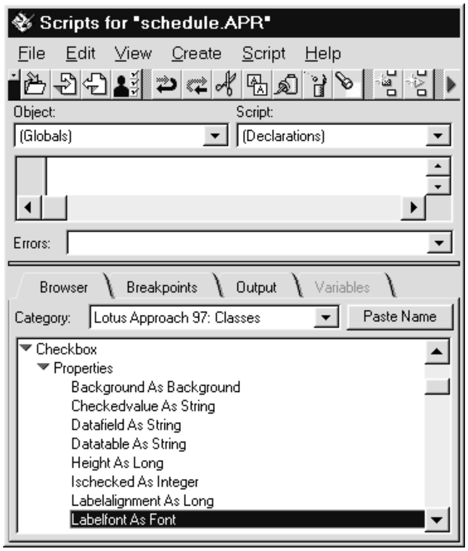
```
Sub Main
```

Errors:

Category: WordPro: Classes

```
▶ Accelerators
▼ Alignment
  ▼ Properties
      Alignmentchar As Integer
      Alignmenttype As Variant
      Application As Wpapplication
      Description As String
      Isvalid As Integer
      Name As String
```

If you look through the Browser in any SmartSuite product, you can examine many containment relationships. For example, Approach has a CheckBox class, which has a property called LabelFont with a data type of Font. This means that the CheckBox class contains the Font class through the LabelFont property.



Object: (Globals)

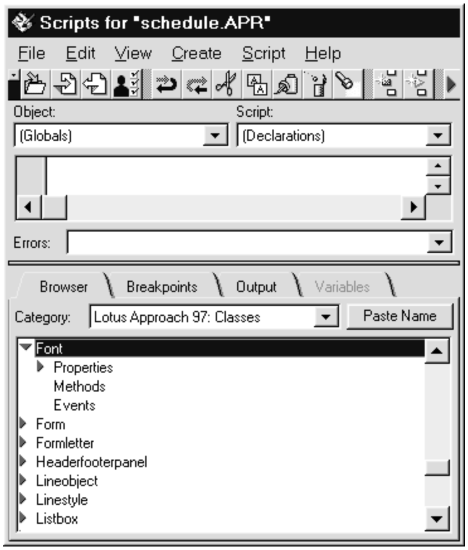
Script: (Declarations)

Errors:

Category: Lotus Approach 97: Classes

- Checkbox
  - Properties
    - Background As Background
    - Checkedvalue As String
    - Datafield As String
    - Datatable As String
    - Height As Long
    - Ischecked As Integer
    - Labelalignment As Long
    - Labelfont As Font

You’ll also notice that Font is listed in the Approach IDE Browser as a class with its own properties.



Object: (Globals)

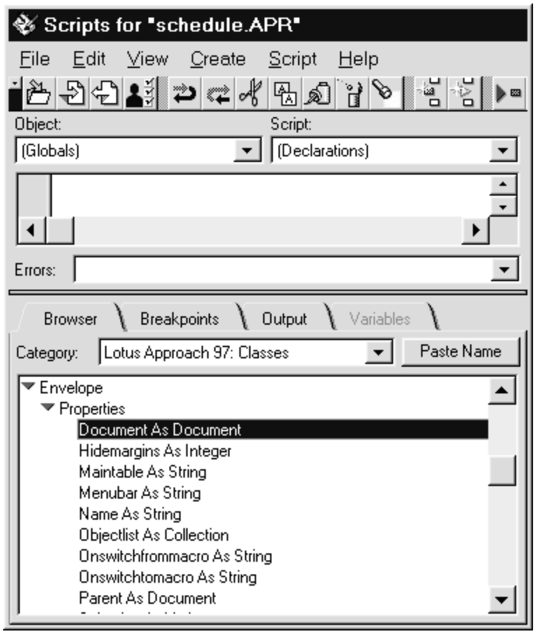
Script: (Declarations)

Errors:

Category: Lotus Approach 97: Classes

- Font
  - Properties
  - Methods
  - Events
- Form
- Formletter
- Headerfooterpanel
- Lineobject
- Linestyle
- Listbox

Notice too that in some cases a class has several properties of the same class data type. For example, the Approach Envelope class has a property called Document with a data type of Document. It has another property called Parent that also has a data type of Document.



Object: (Globals)

Script: (Declarations)

Category: Lotus Approach 97: Classes

- Envelope
  - Properties
    - Document As Document
    - Hidemargins As Integer
    - Maintable As String
    - Menubar As String
    - Name As String
    - Objectlist As Collection
    - Onswitchfrommacro As String
    - Onswitchtomacro As String
    - Parent As Document

Envelope contains Document through the Document property and through the Parent property. This becomes clearer when you think of a property data type such as Integer: the Envelope class has two properties, HideMargins and Visible, that both have a data type of Integer. Document is merely the data type of both the Document and Parent properties, a role it can play because it is contained in the class Envelope.

Inheritance

Inheritance refers to the process by which a class is defined by deriving the members of another class, called the base class. The derived class, sometimes called a subclass, inherits the members (methods, properties, and events) of its base class (sometimes called its superclass). The derived class has direct access to all the members of the base class, but it also has its own new members: additional properties, methods, and events that do not exist for the base class.

The advantage of having a base class is that you can refer to an object of a base class in a script and then use that script for any object of a class derived from the base class, rather than write a different script for each object of a derived class.

Because a derived class inherits all the members of the base class, LotusScript allows you to assign an object of the derived class to a variable with a data type of the base class. However, once the derived class object is assigned to the base class variable, you can access only those derived class members for that object that are shared with the base class. You cannot access any unique members of the derived class.

The following 1-2-3 script is a sub that accepts an argument with the data type of any class derived from the DrawObject class, a base class.

```
Sub ShrinkDrawObject (x As DrawObject)

   x.Height = x.Height / 2
   x.Width = x.Width / 2

End Sub
```

In this sub, the parameter x is declared as having a data type of DrawObject, which specifies that this sub works only on objects that are instances of derived classes of the DrawObject class.

DrawObject is a base class with, among others, the following derived classes: Rectangle, Arc, Picture, Map, Ellipse, DrawLine. Since a derived class inherits the properties of its base class, and Height and Width are members of the DrawObject class, the Height and Width properties in this script are valid for all of the derived classes of the DrawObject class. As a result, this single sub, written for the base class, reduces by half the height and width of any instance of a class derived from DrawObject.

After defining the ShrinkDrawObject sub, you can use it with the derived classes of DrawObject. The following sub is the Click event script for any Rectangle object.

```
Sub Click (x As Rectangle)

   Call ShrinkDrawObject(x)

End Sub
```

This sub runs when the user clicks a rectangle, as indicated by the data type, Rectangle, of the argument. When the script runs, it executes the ShrinkDrawObject sub. Since Rectangle is a derived class of DrawObject, ShrinkDrawObject reduces the size of the rectangle that is clicked.

You can also call ShrinkDrawObject in the Click event scripts of each of the other derived classes of DrawObject. The following sub is the Click event script of the Ellipse object.

```
Sub Click (x As Ellipse)

   Call ShrinkDrawObject (x)

End Sub
```

Abstract classes and inheritance

SmartSuite provides some classes called abstract classes that serve only as base classes for sets of derived classes. You cannot create instances of an abstract class.

For example, the Approach Display class, an abstract class, has the following derived classes: Button, CheckBox, DropDownBox, Ellipse, FieldBox, LineObject, ListBox, OLEObject, Picture, PicturePlus, RadioButton, Rectangle, RoundRect, and TextBox. In Approach you can create instances of these classes, but you cannot create instances of the abstract class, Display. A class from which objects are created is called a concrete class.

Downcasting and inheritance

LotusScript has a feature called downcasting that is useful for writing applications that take advantage of inheritance. Downcasting allows you to successfully assign a source value (a variable, parameter, property, or expression) with a data type of a base class to a target (a variable, parameter, or property) with a data type of a derived class, without generating any run-time errors due to the different data types.

Suppose you have a set of concrete classes that are all derived from an abstract base class, and you create a collection of objects of the different derived classes (you can do this because they all share the same set of members from the base class). By definition, a collection is made up of objects of the same class. An object returned from the collection will be treated as having a data type of the abstract base class. Without downcasting, this wouldn't be very useful because you'd only be able to access the methods, properties, and events of the base class for each object. Downcasting "casts" down the inheritance hierarchy from the abstract, base class to the concrete, derived class, allowing you to access all the derived class members.

To illustrate how this works, consider the inheritance relationship between the following three classes: LotusCheckBox, LotusOptionButton, and LotusControl. LotusCheckBox and LotusOptionButton are derived classes of the base class LotusControl. They inherit all the properties and methods of LotusControl, and they each have their own additional properties and methods. You use these classes to create objects that appear on a dialog box.

LotusDialog, another class, represents a dialog box. It has a property called Controls, which returns a collection of LotusControl objects (the controls on the dialog box represented by the LotusDialog object). Such a collection can include both LotusCheckBox and LotusOptionButton objects. As explained earlier in this chapter, you would use the collection to step through and do things to each object in the collection. Since the objects in the collection are probably not of the same type (it's unlikely that you would have a dialog box with only one kind of control), you must write a script that performs type checking at run time to set properties and execute methods that are specific to each control.

For example, suppose you want to uncheck check boxes on a dialog box and make option buttons visible. The following script iterates through the collection of controls on a dialog box represented by the object Dialog1, and then, depending on the data type of the element, performs an action on the object. Notice that in this example, the Value property belongs to the derived class (LotusCheckBox), while the Visible property belongs to the base class (LotusControl).

```
Dim Control As LotusControl
ForAll Control in Dialog1.Controls
   If TypeName(Control) = "LOTUSCHECKBOX" then
      Dim ChkBox As LotusCheckBox
      Set ChkBox = Control
      ChkBox.Value = 0
   Else
      Control.Visible = True
   End If
End ForAll
```

In the preceding example, if the element being indexed is a LotusCheckBox object, then a variable called ChkBox is declared as having a data type of LotusCheckBox. It is then set to a value of the actual element (Control) in the collection. Remember, since Control is an element of the collection, it has a data type of LotusControl, the base class. Downcasting allows you to assign the object stored in Control to ChkBox, despite the different data types of the two variables. Furthermore, downcasting allows access to the members of a different derived class, LotusOptionButton, if the data type of Control is not a LotusCheckBox.

Notice that the data type of the object is verified before Control is assigned to ChkBox. This data type check has to take place at run time before the assignment; otherwise, if an object of a derived type other than LotusCheckBox is stored in Control, and you try to assign it to ChkBox, which has a data type of LotusCheckBox, an error occurs.

Another approach is to always attempt the assignment and trap the run-time error, if it occurs.

```
Dim Control As LotusControl
ForAll Control in Dialog1.Controls
   On Error Resume Next
   Dim ChkBox As LotusCheckBox
   Set ChkBox = Control
   ChkBox.Value = 0
End ForAll
```

In this second example, even if Control stores a value of a derived type other than LotusCheckBox, the assignment is attempted. If it fails, the error is trapped and processing continues. In the first example, which uses the If... Then... Else statement, the data type of the derived class object is checked first. The assignment of the object to the variable ChkBox is attempted only if the data type is correct. In both cases, if the assignment takes place, an object of the derived class is always valid and the derived class members are always available.

Chapter 3
LotusScript Programming Tools

Lotus provides a powerful set of programming tools for creating and debugging applications in SmartSuite products. If you have developed applications in IBM VisualAge or Microsoft Visual Basic, you will be comfortable with the way LotusScript programming tools support the development process:

- The Integrated Development Environment (IDE) consists of an editor, debugger, class browser, and utilities designed to support designing, writing, and debugging scripts.
- The Dialog Editor is an authoring tool for creating custom dialog boxes.
- The LotusScript Extension (LSX) Toolkit is a separate applications development kit for developing reusable class definitions that work with SmartSuite products and Notes.

This chapter introduces each of these tools and illustrates how you can use them throughout the development process: designing applications, writing the code, running and debugging the applications, and developing reusable code libraries for multiple applications.

Using the IDE

Your primary tool for developing applications is the IDE. Each SmartSuite product supporting LotusScript uses the same IDE. Each document that you create in a SmartSuite product stores three types of information:

- All the product objects that you create in the product, such as paragraphs, tables, ranges, and diagrams
- All the scripts that you write in the IDE for objects in that document
- All custom dialog boxes that you create in the Dialog Editor for that document

Whenever you open a SmartSuite document, all the scripts and dialog boxes stored in that document are read by your SmartSuite product automatically. Although you can export your scripts to compiled LotusScript Object (.LSO) files or plain text LotusScript Script (.LSS) files, you do not have to create or manage separate files for your scripts and dialog boxes.

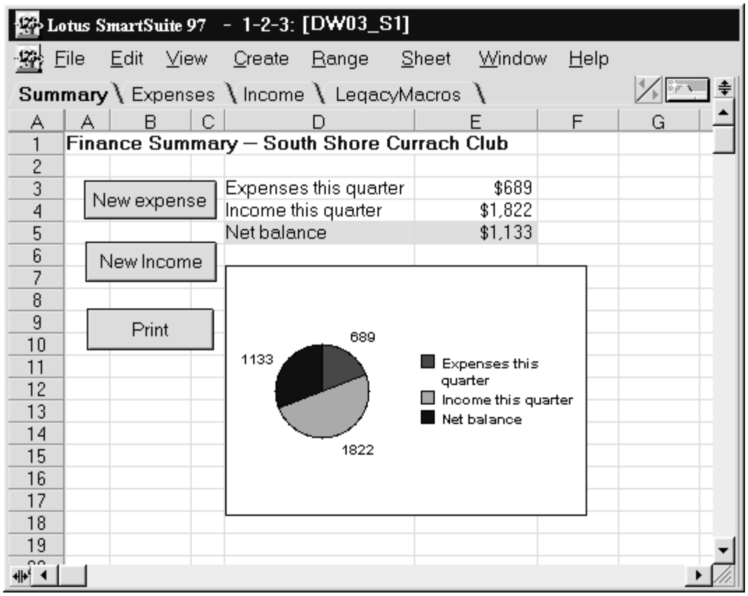
Using the sample application for this chapter

To illustrate how these programming tools support real applications, this chapter provides code examples. All the examples are derived from a sample application named DW03_S1.123 found in the sample files directory. To get the most out of this chapter, load this sample application into 1-2-3 and preview the objects and scripts that it contains.

Opening the IDE

Use the Edit menu command in your SmartSuite product to open the IDE. To open the IDE and view scripts in the sample application for this chapter:

1. Start 1-2-3.
2. Choose File - Open from the 1-2-3 main menu.
3. Select DW03_S1.123 from the directory containing sample applications for this book.



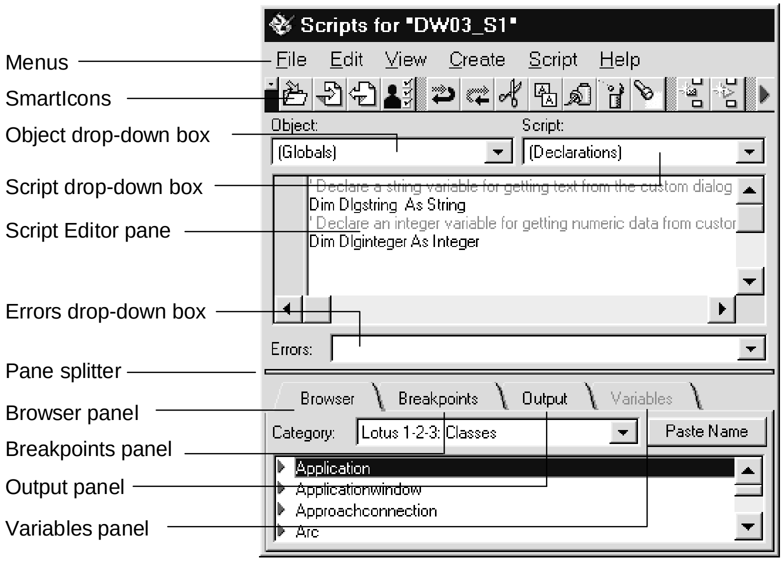
A	A	B	C	D	E	F	G
1	Finance Summary — South Shore Currach Club						
2							
3				Expenses this quarter	$689		
4				Income this quarter	$1,822		
5				Net balance	$1,133		

4. Choose Edit - Scripts & Macros - Show Script Editor to open an IDE window for the current document.

   Note Choose Edit - Show Script Editor in Approach.

Identifying parts of the IDE window

The IDE uses one window for all the programming tools that it provides:



```
' Declare a string variable for getting text from the custom dialog
Dim Dlgstring  As String
' Declare an integer variable for getting numeric data from custor
Dim Dlginteger As Integer
```

- IDE menus and SmartIcons® provide access to commands and preferences.
- The Script Editor or the Script Debugger are displayed alternately in one pane. The Script Editor lets you write scripts and check their syntax. It also lets you set, clear, disable, and enable breakpoints that you use to debug your scripts. The Script Debugger lets you set, clear, disable, and enable breakpoints and step through scripts to locate the source of problems that may occur while a script is executing.
  - The Object drop-down box lists all global and product objects in your current document.
  - The Script drop-down box lists all scripts available for an object selected in the Object drop-down box.
  - The Errors drop-down box lists syntax and run-time errors if they occur.
- The Pane Splitter lets you resize, display, or hide panes in the IDE window.

- The Script Utilities pane has four panels, each containing a tool:
  - The Browser panel lists LotusScript keywords; classes, constants, procedures and variables defined by your product; and type libraries and classes for OLE Automation objects. You can copy items from the Browser panel and paste them into the Script Editor.
  - The Breakpoints panel lists breakpoints that you set in your scripts in the order that you set them, and lets you navigate to, clear, disable, or enable them.
  - The Output panel displays output generated by any LotusScript Print statements that you include in your scripts.
  - During debugging, the Variables panel displays information about variables for the current script and lets you change their values.

Navigating around the parts of the IDE window is easy. To activate a pane or panel, click it or use the View menu commands.

Getting Help in the IDE

You can get several forms of Help while you are working in the IDE:

- Choose Help - LotusScript for reference Help on the LotusScript language and its language elements.
- Choose Help - Script Editor for task-oriented Help on the Script Editor and other programming tools in the IDE.
- Choose Help - *product* Objects for Help on the classes for a particular SmartSuite product.
- Select an entry in any drop-down box and press **F1** to get context-sensitive Help on that entry.
- Move the insertion point to a keyword in a script in the Script Editor or Script Debugger and press **F1** to get context-sensitive Help on that keyword.

Designing applications in the IDE

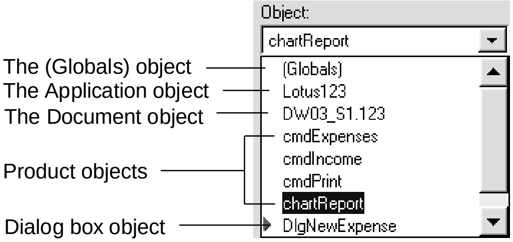
Developing an effective application in LotusScript involves knowing what script resources are available to you and how you can use them in the overall structure of your application. To develop basic applications, you do not need to call external files or write scripts stored outside your current document. To develop more sophisticated applications and groups of applications, you can develop and use external files that contain frequently used scripts and definitions.

Selecting objects for your scripts

LotusScript is closely integrated with the LotusObjects that are created by SmartSuite products. One of the first steps in developing an application in the IDE is determining which objects and accompanying scripts for those objects are relevant to your application.

Using the Object and Script drop-down boxes

The IDE Object drop-down box lists objects that are present in every product document and objects that you create in your document. SmartSuite products create a global object for you named (Globals) in 1-2-3, Approach, and Freelance, or !Globals in Word Pro. The sample application DW03_S1.123 has a default LotusScript object named (Globals), default spreadsheet objects such as the 1-2-3 application and your current document, and user-defined objects such as command buttons, pictures, and charts. To write a script for an object in your document, select the name of that object in the Object drop-down box.

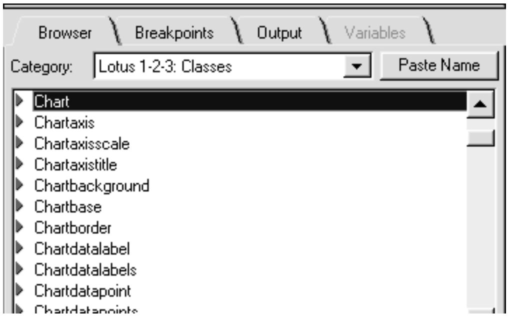


The Script drop-down box lists available scripts for the object selected in the Object drop-down box. Although the scripts available for an object differ according to the scope and function of the object in the product, the following scripts are available for most objects:

- (Options) scripts contain statements that specify LotusScript language options, external .LSO or .LSS files, and some constants used by external files.
- (Declarations) scripts contain declaration statements, constant definitions, and class definitions.
- Initialize scripts set up variables declared in a (Declarations) script.
- Terminate scripts clean up variables declared in a (Declarations) script.
- Event scripts for an object define how that object should respond to particular events that it receives, such as being clicked, moved, or opened.

Using the Browser

The IDE Browser complements the Object and Script drop-down boxes by providing a view into the hierarchy of classes available to you in your product.



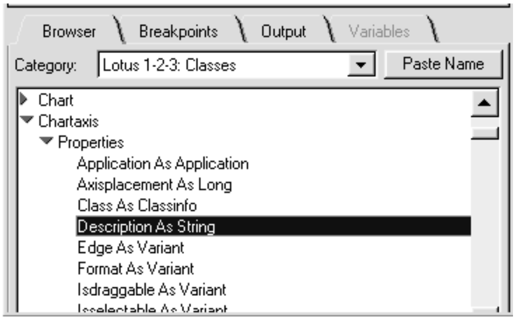
Category: Lotus 1-2-3: Classes

- Chart
- Chartaxis
- Chartaxisscale
- Chartaxistitle
- Chartbackground
- Chartbase
- Chartborder
- Chartdatalabel
- Chartdatalabels
- Chartdatapoint
- Chartdatapoints

The Object drop-down box lists objects currently instantiated in your document (for example a chart named Chart1); the Browser lists classes of objects from which particular objects get instantiated (the Chart class and its derived classes such as ChartAxis or ChartBackground).

Whereas the Script drop-down box lists default event scripts for a product object, the Browser lists all properties, methods, and events associated with each class. Click the right arrow or down arrow to expand or collapse levels of information about the selected entry in the list.

To change the title of a chart axis in your document, for example, you need to know the name of the property that specifies a description for the chart axis title. Select the ChartAxis class in the Browser and expand it to list all its properties, including the one you need, the Description property.



Category: Lotus 1-2-3: Classes

- Chart
- Chartaxis
  - Properties
    - Application As Application
    - Axisplacement As Long
    - Class As Classinfo
    - Description As String
    - Edge As Variant
    - Format As Variant
    - Isdraggable As Variant
    - Isselectable As Variant

Tip Press **F1** to display Help on a selected entry in the Browser.

Recording scripts

Recording scripts from your product is another useful way to learn about the kinds of objects that are relevant to the design of your application. If your product supports this feature, do the following to record scripts:

1. Choose Edit - Scripts & Macros - Record Script.

   Note Choose Edit - Record Transcript in Approach.

2. Specify the location and name of the script into which the IDE records script statements.
3. Click OK to begin recording.
4. Perform actions in your product that are similar to those that you plan for your application.
5. Choose Edit - Scripts & Macros - Stop Recording.
6. Review the recorded scripts in the Script Editor.

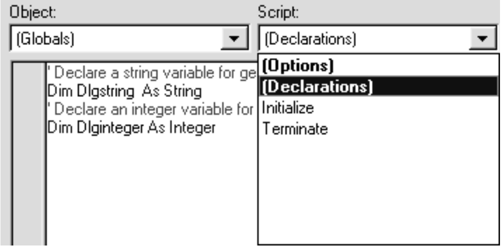
Planning the scope of your scripts

The IDE offers two ways to manage the scope of scripts:

- Write them in (Globals) or in an object script.
- Declare their scope explicitly.

Defining scope in (Globals) and object scripts

Besides product objects, each document has a special object named (Globals) or !Globals. The (Globals) object functions similarly to global modules or .BAS files in other BASIC programming environments. If you have identified declarations and procedures that need to be available to all scripts in your document, you should write them in one of the scripts for the (Globals) object.



Object: (Globals)

Script: (Declarations)

```
' Declare a string variable for ge
Dim DIgstring  As String
' Declare an integer variable for
Dim DIginteger As Integer
```

The IDE automatically adds the following statement to the (Options) script in (Globals):

```
Option Public
```

This has the effect of making all variable, constant, procedure, class, and type declarations that you create in (Globals) public by default.

Unlike scripts in (Globals), object scripts contain variable, constant, procedure, class, and type declarations that are available only to other scripts for that object.

Declaring scope explicitly

You can use the Public and Private keywords to specify the scope of your declarations explicitly. Using Public does not change the scope of variables declared in (Globals) when Option Public is set. Using Private in a (Globals) declaration limits the scope of the declaration to the current object, be that the (Globals) object or scripts for the current object.

Here are some examples of the scope of public and private declarations in (Globals):

Declaration	*Scope*
Dim Var1 As String	Public by default because of Option Public
Public Var2 As String	Public explicitly
Private Var3 As String	Private explicitly

Declarations that you create in object scripts are available only to other scripts for that object; in effect all such declarations are private. You cannot declare public variables in object scripts.

Working with external script files

As you develop applications that use multiple documents containing scripts, you may require some common declarations or procedures. Although you could copy the same scripts to each document that uses them, it is more efficient to store these common scripts in external script files that you can call from multiple documents. There are two types of external script files that you can call from the scripts in your current document.

- .LSS files containing scripts stored as text
- .LSO files containing compiled scripts

For information on external script files, search on “Files” in the IDE Script Editor Help index.

Recycling macros

In planning your application, you don't have to exclude or reimplement many macros that you have written in previous releases of 1-2-3 or Word Pro. For example, to call a 1-2-3 macro named NEWEXPENSE in your LotusScript application, you can include the following statement in one of your scripts:

```
[NEWEXPENSE].MacroRun
```

To convert and run an Ami Pro macro named GETGLOSS.SMM in Word Pro, include the following statement in your Word Pro script:

```
.Application.MacroPlay("C:\AMIPRO\MACROS\GETGLOSS.SMM")
```

Assembling the pieces for an application

To illustrate how these LotusScript resources work together in an application, here is an inventory of the objects and scripts that comprise the sample application DW03_S1.123. This sample is a basic data-entry application; the first sheet provides command buttons for opening data-entry dialog boxes and for printing a summary report.

Resource	*How the resource gets used*
Global scripts	Declarations in (Globals) used by several scripts
	%Include directive for using constants in an external file LSCONST.LSS
Product objects	A command button named cmdExpenses, used to display a dialog box for entering new income items
	A command button named cmdIncome, used to display a dialog box for entering new income items
	Named ranges for database records
	A chart
Macros	Two macros named NEWEXPENSE and NEWINCOME, used to copy output from the custom dialog boxes to database tables named INCOMETABLE and EXPENSETABLE
External script files	Constants in the file LSCONST.LSS included by reference

Writing scripts in the Script Editor

The IDE Script Editor supports four main tasks:

- Selecting objects and their accompanying scripts
- Creating custom subs, functions, and properties
- Entering LotusScript statements in the selected scripts
- Printing, exporting, and importing scripts

Selecting objects and their scripts

To edit scripts in the IDE, you must select an object in the Object drop-down box and then one of its scripts listed in the Script drop-down box.

Selecting global options and declarations

Global options and declarations are stored in the (Declarations) and (Options) scripts for the (Globals) object. To edit global options or declarations, select (Globals) in the Object drop-down box.

The (Options) script in (Globals) in the sample application DW03_S1.123 contains global options for all scripts in the application:

```
' All declarations in (Globals) are public by default.
Option Public
' All variables must be declared explicitly.
Option Declare
' Include the contents of an external script file
' that contains LotusScript constants.
%Include "C:\LOTUS\COMPNENT\LSCONST"
```

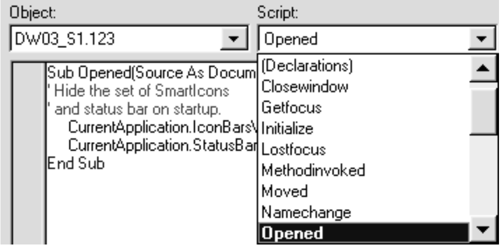
The (Options) script is designed to contain Option statements, Def*type* statements, Use and UseLSX statements, and Const statements needed for Use and UseLSX statements.

To edit global declarations, select (Declarations) in the Script drop-down box. The following global declarations are used by several scripts in DW03_S1.123:

```
' Declare a String variable for getting text
' from the custom dialog box controls.
Dim Dlgstring As String
' Declare an Integer variable for getting numeric
' data from custom dialog box controls.
Dim Dlginteger As Integer
```

Selecting document scripts

You can also select and edit scripts attached to the document itself. To edit a document script, select the name of your document in the Object drop-down box and select a script attached to the document in the Script drop-down box.



Object: DW03_S1.123

Script: Opened

```
Sub Opened(Source As Docum
' Hide the set of SmartIcons
' and status bar on startup.
    CurrentApplication.IconBarsV
    CurrentApplication.StatusBar
End Sub
```

Document scripts are useful when you want to manage what happens when a document gets opened, closed, moved, resized, and so on. The following script from DW03_S1.123 illustrates how to hide elements of the 1-2-3 user interface when the document is opened in 1-2-3.

```
Sub Opened(Source As Document)
   ' Hide the set of SmartIcons and status bar on startup.
   CurrentApplication.IconBarsVisible = False
   CurrentApplication.StatusBarVisible = False
End Sub
```

Selecting scripts for product objects

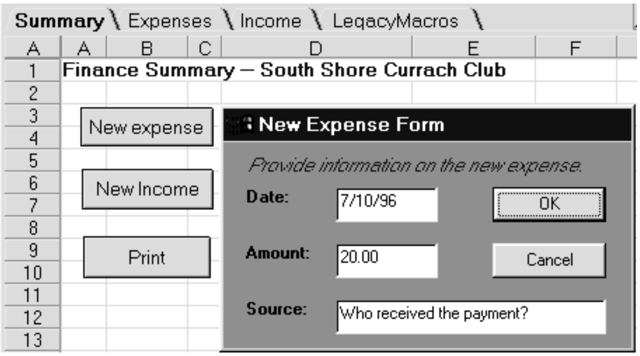
Most of your work in programming SmartSuite products involves creating and writing scripts for product objects. You do not need to create all the scripts for product objects because each object in your document has a set of default scripts. Initially these default scripts are empty. Typically a product object has the following default scripts:

- An (Options) script designed to contain Option statements, Def*type* statements, Use and UseLSX statements, and Const statements needed for Use and UseLSX statements.
- A (Declarations) script designed to contain Dim statements for variables that you want to be available to all scripts for the current object and Const statements for those constants that you want to be available to all scripts for the current object and that are not needed for Use or UseLSX statements in (Options).
- An Initialize sub designed to set up variables you declared in the object’s (Declarations) script.

- A Terminate sub designed to clean up variables that you declared in the object’s (Declarations) script.
- Event scripts designed to manage the way an object responds to events that it receives, such as being clicked, selected, or moved.

For information on working with object scripts, search on “Object scripts” in the IDE Script Editor Help index.

One of the event scripts for the command button named cmdExpenses (labeled “New expense”) in DW03_S1.123 displays a custom dialog box when the user clicks that button.



Finance Summary — South Shore Currach Club

New Expense Form

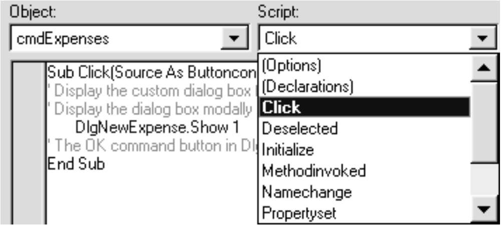
Provide information on the new expense.

Date: 7/10/96

Amount: 20.00

Source: Who received the payment?

To edit this script, select cmdExpenses in the Object drop-down box and select Click in the Script drop-down box.



Object: cmdExpenses

Script: Click

```
Sub Click(Source As Buttoncon
' Display the custom dialog box
' Display the dialog box modally
     DlgNewExpense.Show 1
' The OK command button in Dl
End Sub
```

The following script is attached to the command button and opens the custom dialog box DlgNewExpense.

```
Sub Click(Source As ButtonControl)
   ' Display the custom dialog box for entering
   ' new expense items.
   ' Display the dialog box modally using the argument "1".
   DlgNewExpense.Show 1
End Sub
```

Selecting scripts for custom dialog boxes

A dialog box has two types of scripts, those attached to the dialog box itself and those attached to the dialog box controls that it contains. To select a script for a dialog box, click the name of the dialog box in the Object drop-down box and select one of its scripts in the Script drop-down box. To select a script for a dialog box control, click the right arrow next to the name of the custom dialog box in the Object drop-down list, select the name of one of the dialog box controls in that custom dialog box, and select one of its scripts in the Script drop-down box.

Note The process for developing a custom dialog box in the Dialog Editor is described later in this chapter.

To edit the Click event script for the command button named CmdExpenseCancel in the custom dialog box named DlgNewExpense:

1. Select DlgNewExpense in the Object drop-down box.
2. Press **ENTER** or click the right arrow to display the names of controls in that dialog box.
3. Select CmdExpenseCancel.
4. Click the Script drop-down box to display a list of scripts available for CmdExpenseCancel.
5. Select Click to display the script.

```
Sub Click(Source As LotusCommandButton)
   ' Close the DlgNewExpense dialog box without
   ' processing any of its content.
   DlgNewExpense.Close
End Sub
```

When the user clicks the Cancel button in the dialog box DlgNewExpense, 1-2-3 closes the dialog box.

Creating scripts

You can create your own subs, functions, and properties in (Globals) or for any object that is listed in the Object drop-down box. Once you create these scripts, the IDE lists them in the Script drop-down box. Scripts that you create in (Globals) can be called from any script in your document, unless you declare them explicitly as private. Scripts that you create for objects can be called only from scripts attached to that object.

There are three ways to add scripts to (Globals) or to object scripts:

- Enter the opening line of a Sub, Function, or Property statement in a script (except within a class). When you move the insertion point off the line, the IDE completes the procedure by appending a blank line and the appropriate End line and moves the statements to its own script.

  Note If you entered the opening Sub, Function, or Property line in a script for a product object, the IDE creates the script for that object; otherwise, the IDE moves the script to (Globals).
- Use Create - Sub and Create - Function in the IDE main menu to create new subs and functions. You can specify whether the new script should be added to scripts for a product object or for (Globals).
- Use File - Import Script to add scripts contained in a text file to (Globals) or to object scripts.

In each case, the IDE automatically adds the name of the new script to the Script drop-down box for (Globals) or for an object. To distinguish scripts that you created from the predefined ones, the IDE displays them in bold italics in the drop-down box.

For information on creating scripts, search on “Scripts, creating” in the IDE Script Editor Help index.

Renaming scripts

You can rename subs, functions, and properties in your document; you cannot rename (Options) or (Declarations). To rename a procedure:

1. Navigate to the procedure.
2. Navigate to the first line of the procedure, which includes its name.

   For example:

   ```
   Sub GetNotesMail (TemplateName as String)
   ```
3. Change the name of the procedure.

   For example:

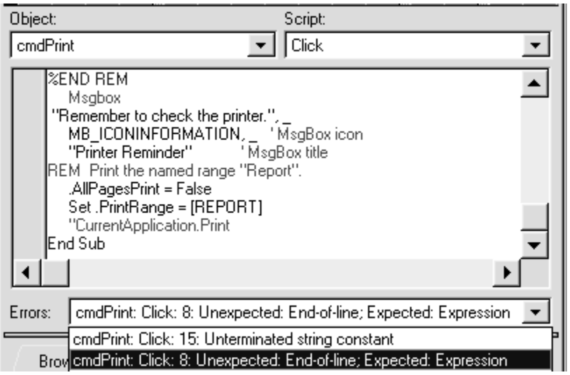
   ```
   Sub GetNotesNetMail (TemplateName as String)
   ```

   Note You cannot have duplicate names in (Globals) or within the scripts for a particular object. If you enter a duplicate name under these circumstances, the Script Editor displays a message notifying you of the error.
4. Move the insertion point off the line. The Script Editor lists the procedure with its new name in the Script drop-down box.

Tip Choose Edit - Find and Replace to replace references to the old procedure name in your existing statements.

Entering text in scripts

The Script Editor provides immediate feedback on the syntax and formatting of text that you enter in scripts. When you complete a line by moving the insertion point to another line, the Script Editor formats the line for correct indentation and checks the line that you typed for syntax errors such as missing argument separators, missing quotation marks, incorrect nesting for arguments, and so on. If it finds a syntax error in a line, it marks the line in a new color and provides information about the error in the Errors drop-down box.



Object: cmdPrint

Script: Click

```
%END REM
   Msgbox
 "Remember to check the printer.", _
   MB_ICONINFORMATION, _   ' MsgBox icon
   "Printer Reminder"          ' MsgBox title
REM  Print the named range "Report".
   .AllPagesPrint = False
   Set .PrintRange = [REPORT]
   "CurrentApplication.Print
End Sub
```

Errors: cmdPrint: Click: 8: Unexpected: End-of-line; Expected: Expression

cmdPrint: Click: 15: Unterminated string constant

cmdPrint: Click: 8: Unexpected: End-of-line; Expected: Expression

To get more information about the syntax error and an example of correct syntax, highlight the error in the drop-down box and press **F1**.

The following example from DW03_S1.123 illustrates how to enter comments, LotusScript language statements, product statements, and multiline statements. The script is attached to the Click event for the 1-2-3 command button named cmdPrint on the first sheet. When the user clicks cmdPrint, 1-2-3 displays a message box, selects a named range for printing, and prints the range.

```
Sub Click(Source As ButtonControl)
   ' Display a message box reminding the user to
   ' check the shared printer.
   %REM
      The constant MB_ICONINFORMATION is derived from
      the external script file LSCONST.LSS.
   %END REM
   MsgBox "Remember to check the printer.", _
      MB_ICONINFORMATION, _                ' MsgBox icon
      "Printer Reminder"                   ' MsgBox title
   Rem Print the named range "Report".
   Set CurrentPrintSettings.PrintSelection = [REPORT]
   CurrentApplication.Print
End Sub
```

Entering comments

You can add comment lines to any script in your document. LotusScript does not execute any text in a comment. LotusScript uses the following conventions for comment lines:

- A single quotation mark designates the beginning of a comment on one line. Although the single quotation mark need not begin the line because it can follow a statement, all text after the single quotation mark is part of the comment.
- The keyword Rem designates the beginning of a comment on one line. Although Rem need not begin the line because it can follow a statement, all text after Rem is part of the comment.
- The %REM and %END REM directives designate the beginning and end of one or more lines of comment. %REM and %END REM must be the first text on a line although they may be preceded on the line by spaces or tabs. Each must be followed by one or more spaces, tabs, or newline characters before any more text appears. %REM blocks cannot be nested.

If you receive a syntax error message on a comment that you enter, press **F1** to get Help on entering legal comments.

Entering LotusScript language statements

When you enter a statement containing a LotusScript language keyword, such as Print or MsgBox in the example above, the Script Editor displays the keyword in a different color to differentiate it from comments, data, or product statements.

For information on the LotusScript language and LotusScript language keywords, search on “Keywords” in the IDE Script Editor Help index.

Entering product statements

The Script Editor also checks the syntax of scripts that contain product statements such as the following:

```
Set CurrentPrintSettings.PrintSelection = [REPORT]
CurrentApplication.Print
```

The Script Editor marks product keywords in a color to distinguish them from language keywords, comments, and data.

Entering multiline statements

The Script Editor does not wrap long lines to fit the width of the IDE window; you can navigate to any part of a long line. For the sake of readability, however, you can break a long statement across multiple lines in the Script Editor. To break a long statement into multiple lines, terminate each line with a space and a line continuation character (_), an underscore.

```
MsgBox "Remember to check the printer.", _
   MB_ICONINFORMATION, _
   "Printer Reminder"
```

The Script Editor does not provide feedback on syntax until you complete the multiline statement.

For more information on multiline statements, search on “Line continuation character” in the IDE Script Editor Help index.

Printing scripts

To print your current script, all scripts for the current object, or all scripts in your document:

1. Choose File **-** Print Script from the IDE main menu.
2. Specify the range of scripts to print:
   - Current script
   - All scripts for current object
   - All scripts for document
3. Click Print to begin printing the scripts.

Exporting scripts to external .LSO or .LSS files

All your scripts and dialog boxes are stored in a product document. If you want to export some or all of the scripts in your document, you can do it in two ways: by exporting scripts in (Globals) to an object file or exporting scripts to a text file.

Exporting global scripts to an .LSO file

.LSO files contain public definitions that you can use in your applications. To export the global scripts in your current document to an .LSO file:

1. Choose File **-** Export Globals as LSO from the IDE main menu.
2. Specify a name and location for the .LSO file.
3. Click Save to create the .LSO file.

Once you create an .LSO file containing public definitions, you can make its definitions available to any of your applications with the Use statement.

```
Use "C:\LOTUS\ADDINS\WKREPORT.LSO"
```

For information on LotusScript Object files, search on “LSO files” in the IDE Script Editor Help index.

Exporting scripts to an .LSS file

You can also export scripts in your current document to an .LSS file. .LSS files contain plain text versions of scripts. You can edit .LSS files in any text editor, import them to other applications, or reference them from applications.

To export the scripts in your current document to an .LSS file, do the following:

1. Choose File - Export Script from the IDE main menu.
2. Specify a scope for the scripts to be exported.
   - Current script only
   - All scripts for this object
3. Specify a name and location for the .LSS file.

   Note The default extension for LotusScript script files is .LSS; you can use any extension or no extension.
4. Click Save to create the .LSS file.

Once you create the .LSS file, you can use its scripts by entering the following statement in your current document:

```
%Include "C:\LOTUS\COMPNENT\LSCONST.LSS"
```

For information on LotusScript script files, search on “LSS files” in the IDE Script Editor Help index.

Running and debugging scripts in the Script Debugger

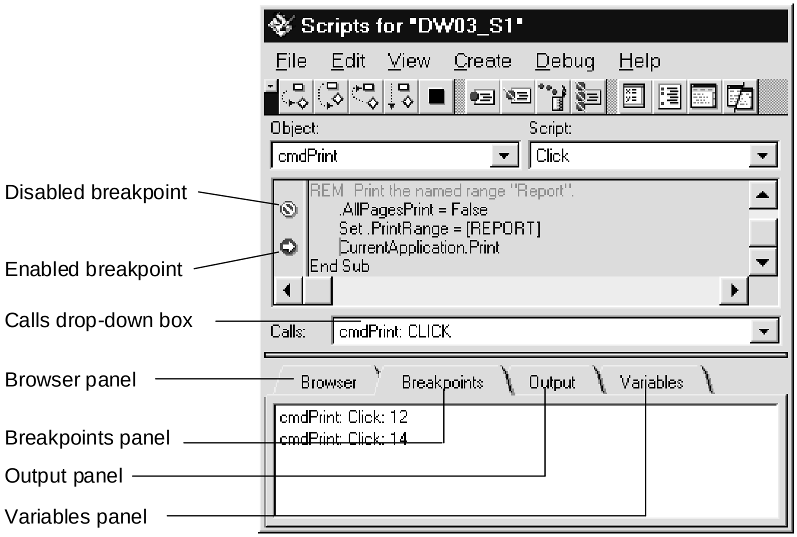
Once you have written your scripts in the Script Editor, the IDE Script Debugger offers a powerful set of tools for running scripts, debugging problems in the scripts, and monitoring how efficiently your scripts execute. In the IDE Script Debugger, you can set, clear, disable, and enable breakpoints and step through scripts to locate the source of problems that may occur while a script is executing.

The Script Debugger

The Script Debugger lets you examine your script as it executes in your product. In the Script Editor, you can set, clear, enable, and disable breakpoints in your scripts. When you run a script containing an enabled breakpoint, the Script Debugger pauses script execution at the breakpoint and lets you evaluate how the script is running.

The Script Debugger is displayed automatically when a script is executing and it encounters one of your breakpoints, a Stop statement in your script, or a run-time error.

The following illustration shows the major parts of the Script Debugger:



Object: cmdPrint

Script: Click

```
REM  Print the named range "Report".
     .AllPagesPrint = False
     Set .PrintRange = [REPORT]
     CurrentApplication.Print
End Sub
```

Calls: cmdPrint: CLICK

```
cmdPrint: Click: 12
cmdPrint: Click: 14
```

Disabled breakpoint

Enabled breakpoint

Calls drop-down box

Browser panel

Breakpoints panel

Output panel

Variables panel

The utilities in the Script Debugger provide additional support for debugging your scripts:

- The Calls drop-down box provides information about the name of your current procedure and any procedures that called it.
- The Breakpoints panel lists all breakpoints that you set in your scripts in the order that you set them, and lets you navigate to them, clear them, enable them, or disable them.
- The Output panel displays output generated by LotusScript Print statements that you put in your scripts.
- The Variables panel displays information about variables for the current procedure and lets you change their values while debugging the script.

Running scripts

You can run scripts while the IDE window is open or closed. Here are some guidelines for running scripts:

- If your current sub has no parameters, you can run it by pressing **F5** or by choosing Script - Run Current Sub.
- If your current sub has parameters that are supplied by the procedure that calls it, you can select that calling procedure and run it.
- If your current sub is an event script for a product object, you must trigger the event for the product object in order to run the sub.

Using breakpoints

Breakpoints are markers that you set in your scripts to assist in the debugging process. When you set a breakpoint on a statement, the Debugger pauses at that line when it executes your script. You can get an accurate picture of how your script is executing by placing breakpoints at key points in your script. If you do not want a breakpoint to pause execution of your script, you can disable it. In effect, the breakpoint is still on the line in case you need it again, but the Script Debugger does not pause execution at that line.

Setting and clearing breakpoints

You can set or clear breakpoints in your scripts with the mouse or with Script or Debug menu commands. The IDE sets breakpoints for individual lines; you cannot set breakpoints for a group of selected statements. You can clear breakpoints for individual statements, for a group of selected statements, or for all statements in your document.

For information on setting and clearing breakpoints, search on “Breakpoints” in the IDE Script Editor Help index.

Enabling and disabling breakpoints

You can enable or disable breakpoints in your scripts with the mouse or with Script or Debug menu commands. Disabling breakpoints is useful when you want to stop using breakpoints temporarily but plan to use them again later.

For information on enabling and disabling breakpoints, search on “Breakpoints” in the IDE Script Editor Help index.

Stepping through scripts

You can identify potential problems with the flow of control or execution of your script by stepping through your scripts on a statement-by-statement or procedure-by-procedure basis.

The following menu items for stepping through scripts are available in the IDE Debug menu and via the Debugger SmartIcons.

Step

Step lets you execute a script procedure one statement at a time. This lets you evaluate the effect of each statement on your product or on your script variables. If a statement in the current procedure calls another procedure, the Script Debugger also executes statements in that procedure one at a time. If another product window was active before the Script Debugger stepped to the current script statement, that product window is activated once again before the Script Debugger executes the next statement in your script. As you step through statements in your script, the Variables panel updates accordingly.

Step Over

Step Over works the same as Step with the exception that Step Over does not stop inside procedures that are called by the current statement.

Step Exit

Step Exit completes execution of the current procedure and stops at the first statement after the current procedure call. From there you can use other Step commands. If the current procedure was called from the product, the script terminates normally. If another product window was active before the Script Debugger executed the current script, that product window is activated once again before script execution resumes.

Continue Execution

Continue Execution resumes script execution until the current script completes successfully or until a breakpoint is encountered. If another product window was active before the Script Debugger resumed execution of your script, that product window is activated once again before script execution resumes.

Stop Execution

Stop Execution terminates the execution of your current script at its current location. The IDE displays the Script Editor.

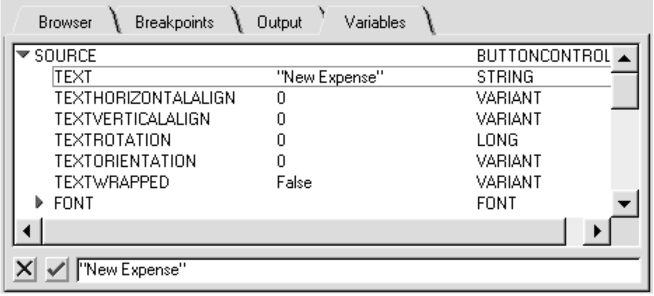
Monitoring variables in your scripts

You can monitor and change the value of a variable when scripts that use that variable are running in the Script Debugger. The Variables panel lists the current value of variables used in scripts that are running. You can monitor how your scripts are using variables by changing their values while the script executes.

To change a variable value during debugging:

1. In the Calls drop-down box in the Script Debugger, select the procedure that contains the value definition.
2. In the Variables panel, select the variable whose value you want to change.

   If the variable value can be changed, the value box, at the bottom of the Variables panel, is enabled. The following illustration shows where you could change the value of the Text property of the command button from "New Expense" to something else.



▼ SOURCE		BUTTONCONTROL
TEXT	"New Expense"	STRING
TEXTHORIZONTALALIGN	0	VARIANT
TEXTVERTICALALIGN	0	VARIANT
TEXTROTATION	0	LONG
TEXTORIENTATION	0	VARIANT
TEXTWRAPPED	False	VARIANT
▶ FONT		FONT

☒ ✓ "New Expense"

3. In the value box, enter a new value for the variable.

   If a variable stores more than 1024 characters, you can view and edit only the first 1024 characters in the box.
4. Press **ENTER** or click the check mark next to the box to change the value of the variable.
5. Press **ESC** or click the check mark next to the box to cancel editing the value of the variable.

Monitoring calls in your scripts

The Calls drop-down box provides a view into calls between procedures, that is, which procedure makes what calls to other procedures as your script executes. The Calls drop-down box has an entry for each call between procedures. Each item contains the name of the current procedure, the name of the procedure that called it, other procedures that called it, objects associated with each procedure, and so on. When you select an item, the

Script Debugger displays the script that made the procedure call. A white arrow in the breakpoint gutter marks the current script statement. A yellow arrow in the breakpoint gutter marks the location of a call to the procedure that contains the current statement.

Items in the drop-down box are dimmed if they are procedures contained in another product document, referenced with the %Include directive, or used with the Use statement. You cannot select these items.

When you select a script in the list, the Variables panel displays variables for the current procedure, their current values, and any global variables used by the script. You can use the Variables panel to view information about a variable or change its value.

Developing custom dialog boxes in the Dialog Editor

The Dialog Editor provides an easy-to-use set of tools for creating custom dialog boxes in 1-2-3, Freelance, and Word Pro. The process is familiar:

- Create the dialog box using the Dialog Editor.
- Add controls to the dialog box using the Dialog Editor.
- Write scripts for the controls using the IDE.
- Run the scripts and display the dialog box in your SmartSuite product.

Note Custom dialog boxes are stored in product documents along with the other scripts in your application.

Creating a custom dialog box

To open the Dialog Editor for your current document, choose Edit - Scripts & Macros - Show Dialog Editor from a product menu.

The Dialog Editor provides a tabbed panel for each dialog box in your current document. To select a dialog box to edit in the Dialog Editor, click its tab.

To create a new dialog box and set some of its design properties:

1. Choose Create - Dialog from the Dialog Editor main menu to create a new dialog box panel in the Dialog Editor.
2. Right-click the background of the new dialog box to select it and to display the shortcut menu for that dialog box.
3. Choose Properties from the shortcut menu to display the InfoBox for that dialog box.

4. Click the Basics tab in the InfoBox to display a panel containing fields for the name and caption of the dialog box.
5. Change properties for the dialog box.

Note Some dialog box properties can be set only at design time: colors, fill patterns, Help file names for context-sensitive Help, and Help topic ID.

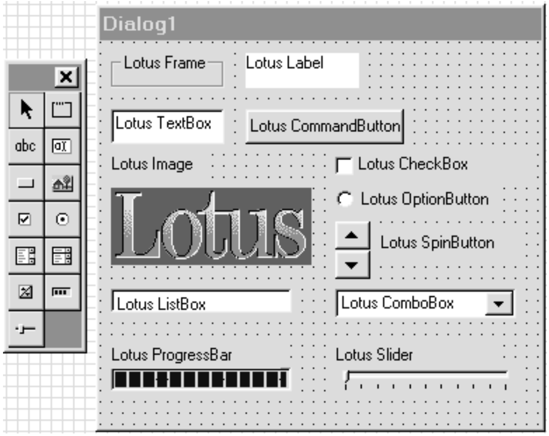
The sample application DW03_S1.123 has two custom dialog boxes, DlgNewExpense and DlgNewIncome. To view the design of the dialog box DlgNewExpense, open the Dialog Editor and click the tab named DlgNewExpense.

Adding controls to the dialog box

The Dialog Editor is an OLE container that supports OLE controls (OCXs) developed by Lotus and other control vendors. The Dialog Editor provides 12 Lotus controls and displays icons for each control in the control Toolbox. The Toolbox displays the name of controls when you move the mouse pointer over Toolbox icons.

Tip Each Lotus control is listed in the Browser.

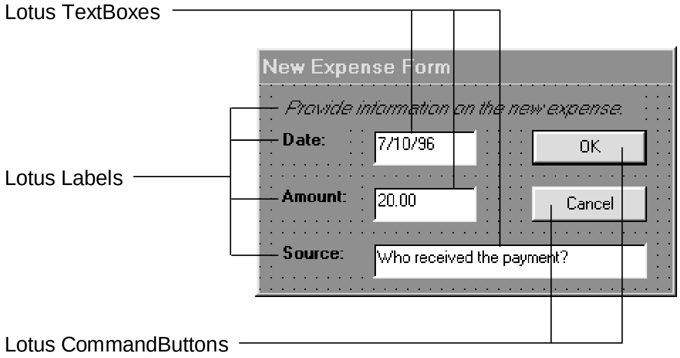
The following illustration shows the Toolbox and the types of controls that you can add to your dialog box.



To add one of these controls to your dialog box, do one of the following:

- Click the icon in the Toolbox, click the background of the dialog box where you want to place the control, and size the control.
- Double-click the icon in the Toolbox to create a default instance of the control in the center of the dialog box and size that control.

The sample dialog box DlgNewExpense uses nine instances of three types of controls:



Lotus TextBoxes

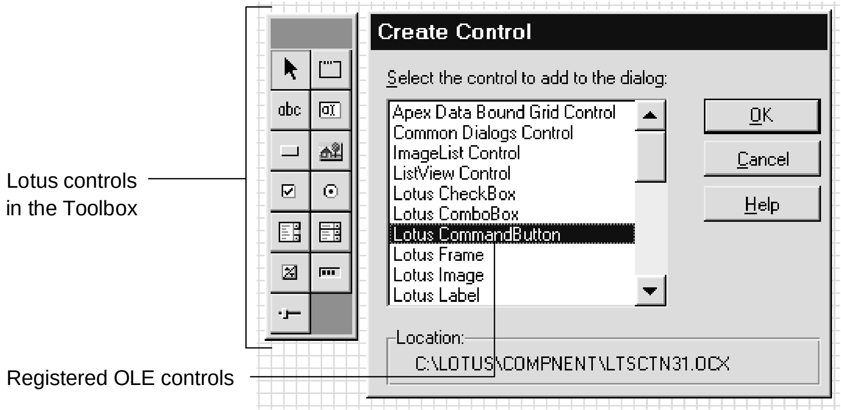
Lotus Labels

Lotus CommandButtons

New Expense Form
Provide information on the new expense.
Date:
7/10/96
Amount:
20.00
Source:
Who received the payment?
OK
Cancel

To examine the properties for a control, right-click the control and choose Properties from the shortcut menu. Use the InfoBox for each control to determine text colors, fonts, borders, names, captions, and default values in list boxes.

To view a list of both Lotus and third-party controls that you can add to your dialog boxes, choose Create - Control - More in the Dialog Editor and select from among the OLE controls registered on your system.



Lotus controls in the Toolbox

Registered OLE controls

Create Control

Select the control to add to the dialog:

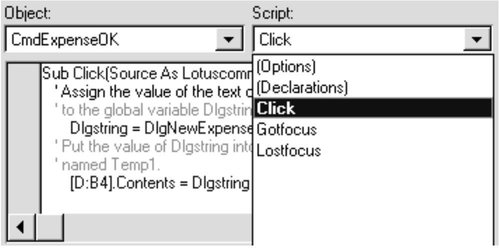
- Apex Data Bound Grid Control
- Common Dialogs Control
- ImageList Control
- ListView Control
- Lotus CheckBox
- Lotus ComboBox
- Lotus CommandButton
- Lotus Frame
- Lotus Image
- Lotus Label

Location: C:\LOTUS\COMPNENT\LTSCTN31.OCX

Writing scripts for controls

The Dialog Editor and the IDE are closely integrated. To display the most recently edited or default script for a control, double-click that control.

The following illustration shows the dialog box control named CmdExpenseOK selected in the Object drop-down box and its corresponding default event script, Click, selected in the Script drop-down box.



Object: CmdExpenseOK

Script: Click

```
Sub Click(Source As Lotuscomn
  'Assign the value of the text c
  ' to the global variable Dlgstrin
    Dlgstring = DlgNewExpense
  ' Put the value of Dlgstring into
  ' named Temp1.
    [D:B4].Contents = Dlgstring
```

As you add controls to your dialog boxes, the IDE adds their names to the list of objects in the Object drop-down box. Click the object in the drop-down list to select it so you can write a script for it. Click the Script drop-down box in the IDE to display a list of scripts associated with the selected object.

When the user enters values or makes selections in a dialog box, you can use that information in your scripts by accessing the properties of the dialog box controls. The following script gets the current value of the text box control named TxtExpenseDate in the dialog box DlgNewExpense and reassigns that value to the sheet cell D:B4.

```
' Assign the value of the text box control TxtExpenseDate
' in DlgNewExpense to the global variable Dlgstring.
Dlgstring = DlgNewExpense.TxtExpenseDate.Text
' Put the value of Dlgstring into cell D:B4, the first
' cell in a temporary range named Temp1.
[D:B4].Contents = Dlgstring
```

Tip If you want to validate input from the user before using it elsewhere in your scripts, you can use the Change event script associated with many dialog box controls. When the user completes entering or selecting information in a control, any scripts in the Change event script are executed.

Running the dialog box from your application

To call a dialog box from your scripts, you must use methods for the Dialog object. The following example illustrates some of the methods for managing dialog boxes at run time.

```
Sub DialogFireDrill
   ' Display DlgNewExpense; run it as a modal dialog
   ' box (that is, assumes control of the product
   ' until the user supplies input).
   DlgNewExpense.Show 1
   ' Close DlgNewExpense.
   DlgNewExpense.Close
   ' Display DlgNewExpense; run it as a modeless dialog
   ' (that is, the user can do other tasks in the product
   ' while the dialog is displayed).
   DlgNewExpense.Show
   ' Hide DlgNewExpense temporarily.
   DlgNewExpense.Hide
   ' Redisplay DlgNewExpense.
   DlgNewExpense.Show
End Sub
```

For more information on using the Dialog Editor and Lotus controls, choose Help from the Dialog Editor main menu.

Developing LotusScript Extension modules

LotusScript Extension (LSX) modules are Dynamic-Link Library (DLL) files that contain public class definitions for any product that supports the LotusScript language. Typically LSX modules add classes to your SmartSuite product, effectively expanding the number of classes available for all your scripts. For example, an LSX module that contains a new class for accessing data in Notes documents would let you connect existing product objects such as 1-2-3 cells or Approach fields to external Notes data.

To make an LSX class available to your scripts, use the following statement:

```
UseLSX "C:\LOTUS\APPROACH\DBENGN01.LSX"
```

Lotus provides LSX modules for Notes and Approach; other LSX modules are being developed for SmartSuite products by Lotus and by third-party developers. To develop your own LSX module, you must obtain a copy of the LotusScript Extension Toolkit. You can download the Toolkit from the World Wide Web by connecting to the Lotus home page (http://www.lotus.com).

Chapter 4
Building a Single-Product Application

The SmartSuite application you develop, whether it uses one or more SmartSuite products, can take advantage of product features and perform tasks that address specific user needs. Any application that you develop is controlled by one or more subs that run inside the product in which the script was written. The subs are saved as part of a document in that product. These scripts act on the objects that the product exposes. In most cases, to run such scripts, you must open the documents in which they are stored, and run them from there.

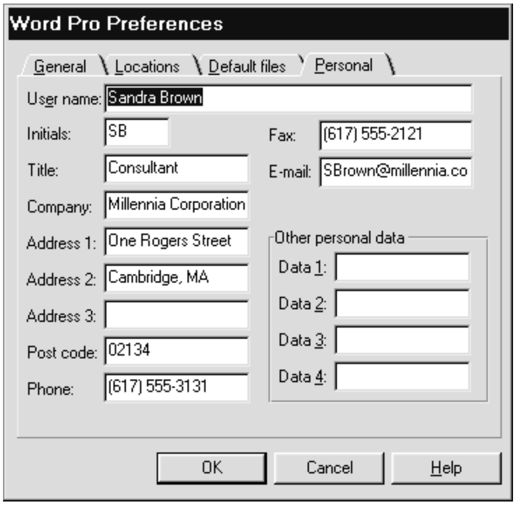
This chapter uses an application developed and run in Word Pro, called the Memo Signing application, to introduce some basic concepts that will help you develop custom applications in SmartSuite products.

Several versions of the application are provided:

- The first version shows a simple script that runs from the Word Pro Edit - Script & Macros menu.
- The second version attaches the same script to one of the icons in the set of SmartIcons so that when the user clicks the icon, the script runs.
- The third version illustrates how to make the script run automatically when the user closes the document in which the script is written.
- The fourth version, made up of several scripts, creates a new menu item, attaches a script to the menu item, and runs the script when the user clicks the menu item.

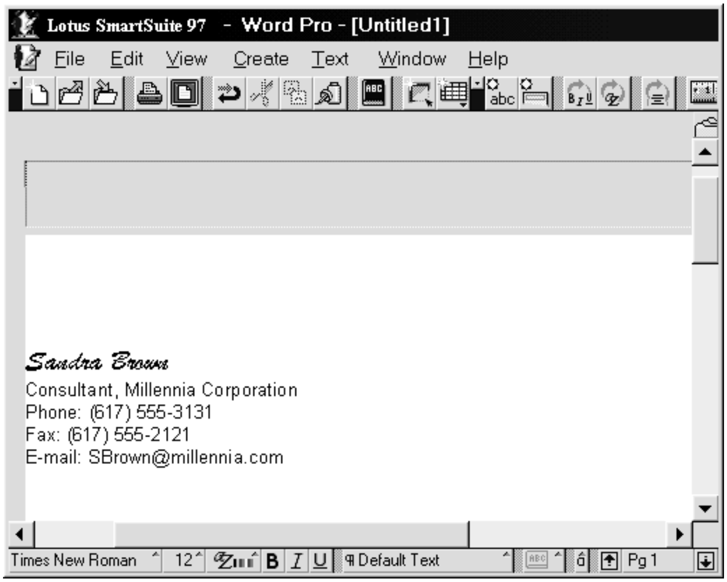
The Memo Signing script

The Memo Signing script automatically adds several lines of information to the end of the current document as a memo or letter signature, using information from the Personal tab in the Word Pro Preferences dialog box.



User name: Sandra Brown
Initials: SB
Title: Consultant
Company: Millennia Corporation
Address 1: One Rogers Street
Address 2: Cambridge, MA
Address 3:
Post code: 02134
Phone: (617) 555-3131
Fax: (617) 555-2121
E-mail: SBrown@millennia.co
Data 1:
Data 2:
Data 3:
Data 4:

When the script runs in a document, it takes the information from the dialog box above, and places the following text at the end of the document.



Sandra Brown
Consultant, Millennia Corporation
Phone: (617) 555-3131
Fax: (617) 555-2121
E-mail: SBrown@millennia.com

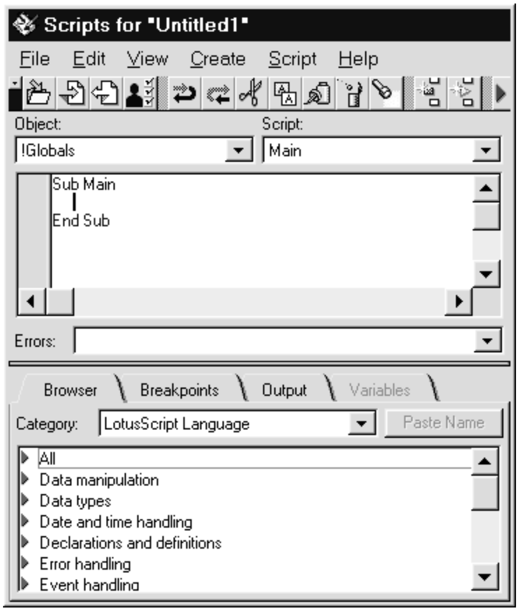
Entering the script

To write the script in this example, do the following in Word Pro:

1. Choose File - New Document to create a new document.
2. Click Create a Plain Document.

   A blank document appears.
3. Choose Edit - Script & Macros - Show Script Editor.

   The Integrated Development Environment (IDE) Script Editor appears with the following text in the Script Editor pane. Notice that the Object drop-down box displays !Globals and the Script drop-down box displays Main.



Object: !Globals

Script: Main

```
Sub Main

End Sub
```

Category: LotusScript Language

4. Type the following script (described in detail in the next section) in the space between Sub Main and End Sub.

   Note The text of this script is stored in DW04_S1.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
Dim Ca As WPApplication
Dim Catxt As Text
Dim Caprf As Preferences

Set Ca = CurrentApplication
Set Catxt = CurrentApplication.Text
Set Caprf = CurrentApplication.Preferences

Call Catxt.MoveToEnd ($LwpLocationTypeDocument)
Call Catxt.InsertBreak ($LwpBreakTypeLine)

Catxt.Font.WindowsName = "Brush Script"
Catxt.Font.Size = 16

Call Ca.Type (Caprf.UserName)

Catxt.Font.WindowsName = "Arial"
Catxt.Font.Size = 10

Call Catxt.InsertBreak ($LwpBreakTypeLine)

If (Caprf.Title <> "") Then
   Call Ca.Type (Caprf.Title + ", ")
End If

Call Ca.Type (Caprf.Company)

If (Caprf.PhoneNumber <> "") Then
  Call Catxt.InsertBreak ($LwpBreakTypeLine)
  Call Ca.Type ("Phone: " + Caprf.PhoneNumber)
End If

If (Caprf.FaxNumber <> "") Then
   Call Catxt.InsertBreak ($LwpBreakTypeLine)
   Call Ca.Type ("Fax: " + Caprf.FaxNumber)
End If

If (Caprf.Email <> "") Then
   Call Catxt.InsertBreak ($LwpBreakTypeLine)
   Call Ca.Type ("E-mail: " + Caprf.Email)
End If
```

5. Choose File - Save As from the Word Pro main menu.
6. Enter the file name SIGNMEMO.LWP in the Save As dialog box.
7. Choose File - Close from the Word Pro main menu to exit the file.

An explanation of the script

This section explains the script in the previous section.

Sub and End Sub

The two lines of code that appear automatically in the Script Editor before you start entering the script are Sub and End Sub, which indicate the beginning and end of a sub. Subs are the parts of a script that perform specific tasks without returning a value (unlike a function, which does return a value). Subs begin with the Sub keyword followed by the name of the sub and the sub arguments, and end with the line End Sub. The script that you write between the Sub and End Sub keywords determines what the sub does.

The sub name, Main, is assigned automatically by Word Pro. Main is the default sub of every Word Pro application: the first time you open the Script Editor, the window appears with an empty sub, Main, in which you can enter code. Main provides a place for entering code that is not associated with an object (usually, you enter code into an event script of an object).

Note You can also use Main to write code that you want to test. For example, if you are writing a script for the Opened event of a Word Pro document, it is more convenient to run the script directly from Main, rather than close and open the document to cause the event to occur and run the event script. In this case, after you run the script from Main and verify that there are no errors, you can copy it into the Opened sub of the Document object.

Setting up the document to sign the memo

The first three lines of the script declare three variables. When you declare a variable, you specify its data type and in some instances its value. It’s good practice to put all declarations at the beginning of a script. That way you always know where to look for information about variables when reading the code. Use the Dim statement, or one of its variations, to explicitly declare a variable type. For more information about declaring variables, search on “LotusScript” in your product Help index, then click “LotusScript Index.”

```
Dim Ca As WPApplication
Dim Catxt As Text
Dim Caprf As Preferences
```

The three variables declared in this script are Ca, Catxt, and Caprf.

- Ca is used to store a WPApplication object, which represents the current Word Pro session.
- Catxt is used to store a Text object, as indicated by the data type of Text. The Text object represents the text in a document.
- Caprf is used to store a Preferences object, as indicated by the data type of Preferences. The Preferences object represents the Word Pro Preferences dialog box.

Next, the script sets the three variables to the objects you want to store in them.

```
Set Ca = CurrentApplication
Set Catxt = CurrentApplication.Text
Set Caprf = CurrentApplication.Preferences
```

CurrentApplication is a predefined global product variable that stores an instance of the WPApplication class. WPApplication contains the Text and Preferences classes as the Text and Preferences properties. By setting Catxt and Caprf to these two contained objects and Ca to CurrentApplication, you make the script easier to write and read, since the variable names are much shorter than the full object specifications.

Next, the script uses the MoveToEnd method of the Text class to move to the end of a line of text or a document. The method argument, $LwpLocationTypeDocument, a Word Pro-specific constant, specifies the location, in this case the end of the document.

```
Call Catxt.MoveToEnd ($LwpLocationTypeDocument)
```

The next line of code uses the InsertBreak method of the Text class to insert a break in the text. The argument is a Word Pro constant, $LwpBreakTypeLine, that indicates the kind of break you want to insert: in this case, a line break, which is equivalent to a carriage return.

```
Call Catxt.InsertBreak ($LwpBreakTypeLine)
```

Printing the user's signature

To print the user's name from the Word Pro Preferences dialog box, the script first sets the font of the text to Brush Script. Notice the use of containment in this line. Remember that Catxt represents CurrentApplication.Text. Text, contained by CurrentApplication, is a property of the WPApplication class and contains a Text object. Font, contained by Text, is a property of the Text class and contains a Font object. WindowsName is a property of the Font class that defines the font name.

```
Catxt.Font.WindowsName = "Brush Script"
```

The size of the text is defined as 16 points by setting the Size property for the Font object.

```
Catxt.Font.Size = 16
```

The user's name, as it appears in the Word Pro Preferences dialog box, is then printed on the screen using the Type method of CurrrentApplication. The argument for the Type method is the value to be printed on the screen. In this case, the value is the variable in which the user's name is stored, Caprf.UserName.

```
Call Ca.Type (Caprf.UserName)
```

Changing the font

To print the rest of the information, the script changes the font from 16-point Brush Script to 10-point Arial and inserts a carriage return.

```
Catxt.Font.WindowsName = "Arial"
Catxt.Font.Size = 10
Call Catxt.InsertBreak ($LwpBreakTypeLine)
```

Printing additional information conditionally

The remainder of the script prints the user's title, company name, phone number, fax number, and e-mail address, if each of these exists on the Personal panel of the Word Pro Preferences dialog box.

For example, the user's title, followed by a comma and a space, is printed if the Title property of the Preferences object contains a value.

```
If (Caprf.Title <> "") Then
   Call Ca.Type (Caprf.Title + ", ")
End If
```

The remaining If statements work in the same manner, printing values if they are stored in properties of the Preferences object.

Running the Memo Signing script

After you write a script, you have to decide how you want the user to run it. Ask yourself the following questions:

- Do you want the user to run the script from the Edit - Script & Macros menu?
- Do you want to attach the script to an icon that the user must click to run the script?
- Do you want the script to run automatically, using an event, when the user opens a Word Pro document?
- Do you want to attach the script to a menu item that the user must click to run the script?

Running the script from the Word Pro Edit - Script & Macros menu

The simplest way for the user to run a script is directly from the Word Pro Edit - Script & Macros menu:

1. In Word Pro, the user chooses Edit - Script & Macros - Run.
2. If the script is stored in the current file, the user selects “Run script saved in the current file” and selects the name of the script from the drop-down box.
3. If the script is stored in another file, the user selects “Run script saved in another file” and enters the path for the file.
4. The user clicks OK.

Running the script from an icon

You probably want to provide your users with a user-friendly interface. Instead of forcing them to go to the Edit pull-down menu and then the Script & Macros cascade from which they choose Run, you can provide them with a more straightforward way to run the script, such as clicking an icon in the set of SmartIcons.

Attaching the script to an icon

After you write the script, follow these steps in any Word Pro document to attach the script to an icon:

1. Choose File - User Setup - SmartIcons Setup.

   The SmartIcons Setup dialog box appears.
2. Click Edit Icon.
3. Click the desired icon in the “Available icons you can edit or copy” box and click Attach Script.

   The Word Pro - Choose Script dialog box appears.

4. Choose the name of the document you just created, SIGNMEMO.LWP. This file contains the script for the Memo Signing application.
5. Click Open to return to the Edit SmartIcons dialog box.

   If you want a description to appear in the icon bubble help, type it in the “Description” box.
6. Click Save.
7. Click Done to return to the SmartIcons Setup dialog box.
8. Click OK.

Running the script

To run the script that is now attached to the icon, simply add the icon to the icon bar. The user can then open a new document (or an existing one, if the user wants to add the signature text to it), and click the icon. The script runs and adds the text to the end of the open document.

Running the script automatically using events

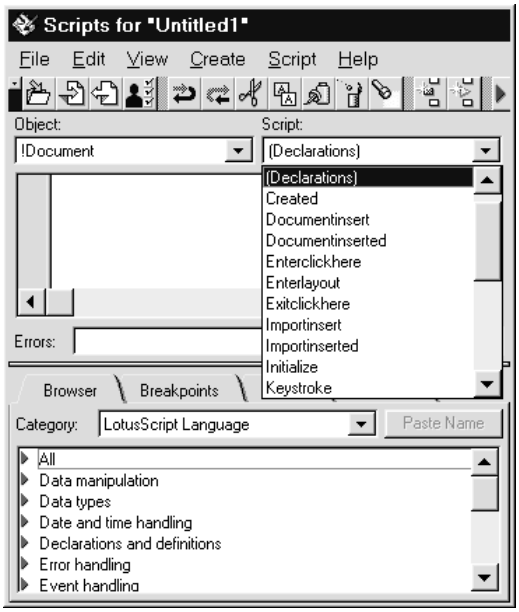
You may decide that the script should run automatically as part of a user’s regular work process. In this case, events are very useful. An event occurs when the user, application, or system, performs an action. Different actions trigger different events. For example, opening a document triggers the Opened event; closing a document triggers the PreClose event. Document/Opened is an object/event pair. Document/PreClose is another object/event pair. If a script is associated with the object/event pair, the script runs automatically when the event is triggered but the user, application, or system.

You can view all the events of an object in the IDE Browser by clicking the name of the class from which that object is instantiated and then clicking the Events subheading under that. For information about viewing class members in the IDE Browser, see Chapter 3.

You can also view the events associated with an object in the Script Editor:

1. Click the Object drop-down box and select an object name, for example !Document, which represents the current document in Word Pro.

2. Click the Script drop-down box for a list of events for the object you selected.



You can enter a script in the Script Editor for each of these events.

Running the script from the PreClose event

The purpose of the Memo Signing script is to sign a document. Most likely, users would want to do this when they are done writing the document. If you attach the script to the PreClose event of the document, the script will run right before the document closes. In other words, the user chooses File - Close, the script runs, and then the document closes.

To enter the script for the PreClose event for the document, open the file called SIGNMEMO.LWP (which you created earlier in this chapter) and do the following:

1. Choose Edit - Script & Macros - Show Script Editor.

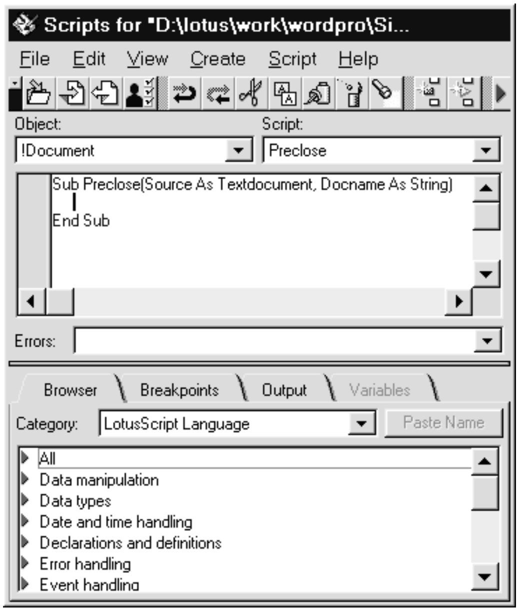
   The IDE Script Editor appears.

2. Click the Object drop-down box and select !Document.

   This is the name that Word Pro automatically assigned to the Document object that represents the document you are working in.

3. Click the Script drop-down box and select PreClose.

The following text appears in the Script Editor. This is the PreClose event sub, with space between the Sub and End Sub keywords for you to write a script for the event.



Scripts for "D:\lotus\work\wordpro\Si...

Object: !Document

Script: Preclose

```
Sub Preclose(Source As Textdocument, Docname As String)

End Sub
```

Errors:

Category: LotusScript Language

4. Type the following statements in the space between the Sub and End Sub keywords:

```
Call Main
Call Source.Save
```

The first statement runs the Main sub, which contains the Memo Signing script, whenever the user closes the document.

The second statement invokes the Save method of the Document object. Notice that Source As TextDocument is one of the parameter declarations of the PreClose sub. The parameter Source, which represents the document being closed, is a convention that SmartSuite products use within an event script to identify the object of the current object/event pair.

Because these statements are in the PreClose event script, Main runs, the signature text is added, and the document is saved, all before the document closes.

5. Choose File - Save.

Now, close the document. The next time the document is opened and closed, the script runs, adds the signature information to the end of the document, and saves the file.

Running the script from a new menu item

Another way to run the script is to attach it to a new menu item so that when the user clicks the menu item, the script runs. For this to happen, the application must do the following:

- Add a new menu item to the Word Pro menu bar that appears when the user opens the document.
- Attach the Memo Signing script to the menu item.
- Run the Memo Signing script.
- Delete the new menu item from the Word Pro menu bar when the user closes the document.

These tasks are performed by three subs (in the file DW04_S2.LWP in the sample files directory) that together form the application: Opened, SignMemo, and PreClose. Opened and PreClose are predefined event subs of the Document object. SignMemo is a user-defined sub that runs when the user clicks the new menu item called Sign.

Opened

The Opened sub is a script associated with the Opened event of the Document object that represents the file DW04_S2.LWP.

Every time the user opens DW04_S2.LWP, the Opened event is triggered and the following script attached to it runs. The script creates a new menu item on the Word Pro menu bar.

```
Sub Opened (Source As TextDocument, DocName As String)

   Dim MenBar As MenuItem
   Dim MenItem As MenuItem

   Set MenBar = _
     CurrentApplication.ApplicationWindow.LwpMenuBar
   Set MenItem = MenBar.NewItem("&Sign", "!SignMemo")

End Sub
```

The Opened sub takes two arguments, Source and DocName. Source represents the Document object. DocName represents the name of the document.

The first two lines of the script declare two variables, MenBar and MenItem. MenBar is used to store the Word Pro menu bar, so it is declared with a data type of MenuItem. MenItem is used to store the new menu item that will be added to the menu bar and is also declared with a data type of

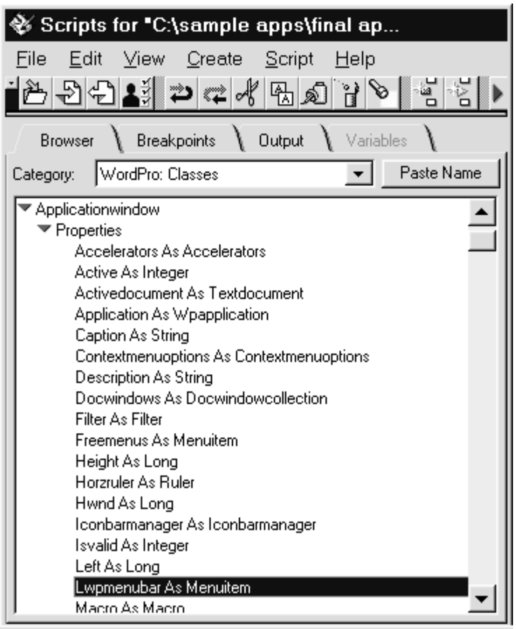
MenuItem. This means that the only kind of data that can be stored in MenBar and MenItem is an instance of the Word Pro MenuItem class. A Word Pro MenuItem object represents either a menu bar or a menu item on a menu bar.

The last two lines of the script are two Set statements. The Set statement initializes a variable to an object.

```
Set MenBar=CurrentApplication.ApplicationWindow.LwpMenuBar
```

The first Set statement, above, sets MenBar to the Word Pro menu bar. CurrentApplication, an object that is an instance of the Word Pro Application class, represents the current Word Pro session. It contains the ApplicationWindow class through the ApplicationWindow property, which represents the Word Pro application window. LwpMenuBar is a property of ApplicationWindow, and represents the Word Pro menu bar.

If you look in the IDE Browser at the list of Word Pro classes and click ApplicationWindow and then Properties, you see LwpMenuBar listed as a property with a data type of MenuItem, indicating that the ApplicationWindow class contains the MenuItem class through the LwpMenuBar property. Since MenBar and LwpMenuBar have the same data type, MenuItem, MenBar can be set to the value of the LwpMenuBar property.



Category: WordPro: Classes

Applicationwindow
Properties
Accelerators As Accelerators
Active As Integer
Activedocument As Textdocument
Application As Wpapplication
Caption As String
Contextmenuoptions As Contextmenuoptions
Description As String
Docwindows As Docwindowcollection
Filter As Filter
Freemenus As Menuitem
Height As Long
Horzruler As Ruler
Hwnd As Long
Iconbarmanager As Iconbarmanager
Isvalid As Integer
Left As Long
Lwpmenubar As Menuitem
Macro As Macro

The second Set statement, next, does several things simultaneously.

```
Set MenItem = MenBar.NewItem("&Sign", "!SignMemo")
```

The statement does the following:

- Creates a new menu item on the Word Pro menu bar using the NewItem method of the MenuItem class.
- Assigns the name Sign to the new menu item and underscores the first character (the letter *S*) of the name.
- Attaches a sub called SignMemo to the new menu item, using "!SignMemo" as an argument of the NewItem method. The exclamation point (!) is a Word Pro script convention.
- Sets the variable MenItem to the new menu item called Sign.

SignMemo

The SignMemo sub is the script now attached to the new menu item. It contains the same code you entered in the Main sub of SIGNMEMO.LWP, earlier in this chapter. The SignMemo sub, triggered when the user clicks the menu item, causes the information from the Word Pro Preferences dialog box to appear at the end of the document.

This sub is a !Globals object script. (!Globals is listed as (Globals) in other SmartSuite products.) For information about defining scope in !Globals scripts, see Chapter 3.

PreClose

The PreClose sub is the script that runs when the Document object PreClose event is triggered. When the user chooses a Word Pro command to close the document (for example, File - Close or File - Exit), the script runs immediately before the document closes.

The script deletes the Sign menu item from the menu bar for two reasons. First, this menu item is associated with this document, but if the menu item is not removed, it will remain on the menu bar when a different document is opened, causing problems when the user clicks the item. Second, the next time this document is opened, another instance of the Sign menu item will be added, which means that the menu bar will contain as many Sign menu items as the number of times the document is opened.

```
Sub PreClose (Source As TextDocument, DocName As String)

   Dim MenBar As MenuItem
   Set MenBar = _
     CurrentApplication.ApplicationWindow.LwpMenuBar
   Call MenBar.DeleteItem ("&Sign")

End Sub
```

The PreClose sub, like the Opened sub, takes two arguments, Source and DocName.

The script declares a single variable, MenBar, with a data type of MenuItem. This variable is local to this sub, and although it has the same name as the variable declared in the Opened sub, it can only be used inside this sub.

The second line in the script sets the variable MenBar to the Word Pro menu bar, as was done in the Opened sub. The last line of the script deletes the menu item called Sign from the Word Pro menu bar using the DeleteItem method of the MenuItem class.

Chapter 5
Building Cross-Product Applications

A major benefit of building applications in SmartSuite is that you can combine data and functionality from multiple products to create a single custom application. Because SmartSuite products that support LotusScript share similar object models and are OLE applications, you can expose LotusObjects from one product to another as OLE Automation objects. These same products are also OLE Automation controllers, so you can use them or any other OLE Automation controller, such as Visual Basic, to manipulate LotusObjects.

This chapter presents an overview of OLE Automation concepts and two sample applications. The first application uses LotusScript to perform OLE Automation between Word Pro and Approach. The second application uses Visual Basic to control objects in 1-2-3.

OLE Automation concepts

OLE Automation lets you access and control one product's objects from a script running in another product. The product through which you manipulate such objects is called the OLE Automation controller. The objects exposed to the OLE Automation controller are called OLE Automation objects. 1-2-3, Approach, Freelance Graphics, and Word Pro function as OLE Automation controllers and also expose all of their LotusObjects as OLE Automation objects.

OLE Automation controllers

LotusScript lets you manipulate LotusObjects in the product in which you are running LotusScript. In addition, you can use LotusScript to control objects in products other than the one in which you are running LotusScript. For example, through LotusScript, 1-2-3 can act as an OLE Automation controller and manipulate objects in Word Pro. All Lotus products that support LotusScript are OLE Automation controllers.

OLE Automation objects

When an OLE Automation controller accesses objects created by another product, the objects that are exposed to the controller are called OLE Automation objects. SmartSuite products provide OLE Automation objects that are the same LotusObjects available in the product in which they are instantiated. The only difference is that they are manipulated by an outside product, the OLE Automation controller.

The OLE Automation controller can manipulate or extract data from the objects in another product by getting and setting properties and invoking methods. Every Lotus product that supports LotusScript can expose its objects as OLE Automation objects. You can also access such OLE Automation objects through Visual Basic, Visual C++™, or other languages written for applications that support OLE Automation.

Accessing LotusObjects using OLE Automation

Once you know how to write scripts that run independently within products, you don't have to learn much more to take advantage of OLE Automation. When you write a script that controls another product's objects, the syntax you use and the methods and properties of those objects are quite similar to what you would use in a script written for that product.

The following script, written in LotusScript to run in Word Pro, is an example of a product manipulating its own objects. This script creates a new Word Pro document, inserts some text, and saves the file:

```
Dim App As WPApplication

Set App = CurrentApplication

Call App.NewDocument ("Foo.Lwp")
Call App.Text.InsertText ("This is a New Line")
Call App.Save
```

You can do the same task from outside Word Pro using a product that runs a scripting language other than LotusScript. The next script is an example of a product manipulating objects in another product, using Visual Basic. The product running the Visual Basic script is an OLE Automation controller and Word Pro is an OLE Automation object:

```
Dim App As Object

Set App = CreateObject ("WordPro.Application")
App.Visible = True

Call App.NewDocument ("Foo.Lwp")
Call App.Text.InsertText ("This is a New Line")
Call App.Save
```

In this example, Visual Basic starts Word Pro, makes it visible, and instructs it to create the new Word Pro document, insert some text, and save the file. The syntax of the Visual Basic example differs from the Word Pro example in the following ways:

- The variable App is declared with the data type of Object, rather than WPApplication. Object is a Visual Basic data type.
- The Visual Basic CreateObject function is used to access Word Pro via OLE Automation.
- The Visible property of the object must be set to True to allow the user to see the Word Pro session.

Notice that aside from the Set statement that accesses Word Pro, the rest of the code in the two examples is the same. The Set statements differ because in the first example, LotusScript is already running in Word Pro; in the second example, Visual Basic is running outside of Word Pro and must use OLE Automation to start the product. The three Call statements in the two examples, however, are identical.

LotusScript applications as OLE Automation controllers

Any product that supports LotusScript can act as an OLE Automation controller. The LotusScript CreateObject and GetObject functions allow you to write scripts that access OLE Automation objects. These objects are stored in variables of type Variant, a LotusScript data type. For information about LotusScript data types, search on “LotusScript” in your product Help index and click “LotusScript Index.”

The following script is an example of one SmartSuite product manipulating objects in another SmartSuite product, using LotusScript. The script can be run, for example, in Approach or in any other product that supports LotusScript. Like the two preceding examples, it creates a new Word Pro document, inserts some text, and saves the file.

In this example Approach is an automation controller accessing Word Pro, an automation object:

```
Dim App As Variant

Set App = CreateObject ("WordPro.Application")
App.Visible = True

Call App.NewDocument ("Foo.Lwp")
Call App.Text.InsertText ("This is a New Line")
Call App.Save
```

Notice that this example closely resembles the Visual Basic example shown in the preceding section. The Word Pro OLE Automation object is created in both examples using the CreateObject method. The only difference is that LotusScript declares the variable App, in which the Word Pro OLE Automation object is stored as type Variant, rather than as type Object.

Variables for storing OLE Automation objects

One of the differences between a LotusObject and a LotusObject accessed via OLE Automation is the data type of the variable in which the object is stored.

- A LotusObject is stored in a variable with a data type of the class of the object.

  For example, CurrentApplication is a predefined global Word Pro variable that stores an object of the WPApplication class. Any variable you set to CurrentApplication must have a data type of WPApplication.

- A LotusObject accessed via OLE Automation is stored in a variable with a LotusScript data type of Variant or a Visual Basic data type of Object.

  For example, suppose you want to automate Word Pro using LotusScript. As in the previous example, you create the OLE Automation object that is a WPApplication object using the CreateObject function and set it to a variable, App, with a data type of Variant.

Storing an OLE Automation object in a variable with a data type of Variant does not affect the availability of the methods and properties of the LotusObject it represents. In the previous example, CreateObject is using OLE Automation to create an instance of the WPApplication class. All the methods and properties of the object stored in App, which is a variable of type Variant, are available to the script, irrespective of OLE Automation.

Object names for applications

All SmartSuite object models have an Application object. From an Application object, you can traverse the hierarchy to find all other objects. To create an OLE Automation object in a Lotus product that has LotusObjects, use the CreateObject method of your scripting language.

Lotus products use the following object names when exposing their objects for OLE Automation:

- Lotus123.Workbook

  Note Unlike the other SmartSuite products, 1-2-3 returns an object of type Document, instead of an object of type Application. To access the 1-2-3 Application object, once you have accessed the Document object, use the Parent property of the Document object (Document.Parent).
- Approach.Application
- Freelance.Application
- WordPro.Application

OLE Automation using LotusScript with Word Pro and Approach

The Video Summary application allows the user to create a document in Word Pro by extracting data from an Approach database. Word Pro, the product in which all the application's scripts are running, is the OLE Automation controller. Approach, the product from which the controller is accessing objects, provides the OLE Automation objects.

In this sample application, Millennia is a video tape supplier that sells videos to stores. A user, perhaps a salesperson at Millennia, might run the Video Summary application every week to generate sales reports for management. When the application runs, the result is a memo containing text and a table filled with sales data that has been automatically extracted from Approach.

Note Before you run this application, make sure that the file DW05_S1.APR is stored on the sample files directory (<*DRIVE*>:\...\SAMPLES\SUITE). If you have moved it to another directory, then follow these steps:

1. Open the Word Pro document DW05_S1.LWP.
2. Choose Edit - Script & Macros - Show Script Editor.
3. Click the Object drop-down box and select !Globals.
4. Click the Script drop-down box and select (Declarations).
5. In the following statement, specify the path (in <*PATH*>) in which the file DW05_S1.APR is stored:

   ```
   Const DBPATH = <PATH>
   ```
6. Click the Script drop-down box and select Menu_Invoices.

7. Uncomment the following statement:

```
' Call gApproach.OpenDocument ("dw05_s1.apr", DBPATH, _
' "" ,"" ,False , True)
```

8. Comment the following statement:

```
Call gApproach.OpenDocument ("dw05_s1.apr", AutoPATH, _
    "" ,"" ,False , True)
```

9. Choose File - Save.
10. Choose File - Close.

Generating a sales report

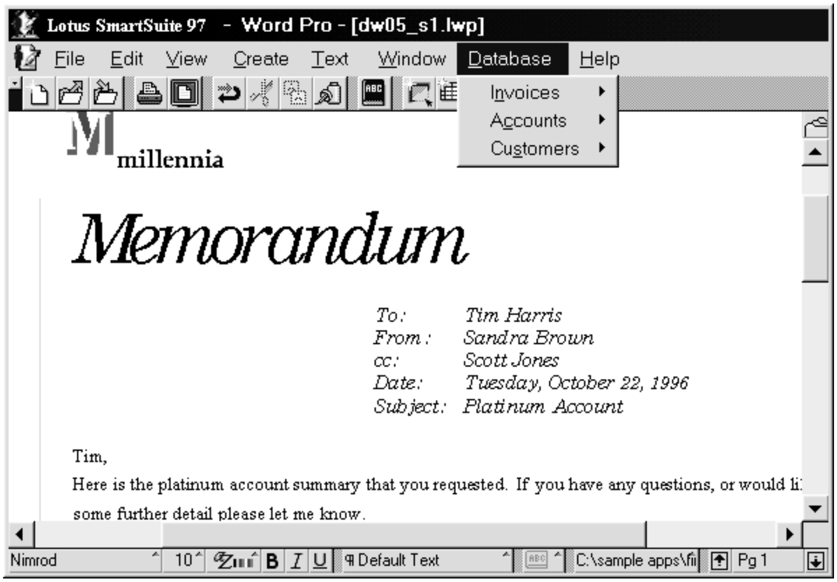
The following steps outline a procedure that the user follows to run the Video Summary application.

1. Open the Word Pro document DW05_S1.LWP.

   A custom menu item, Database, appears in the menu bar. The document initially contains boilerplate text for a memo to the sales manager. As yet, there is no table, no data, in the memo.

2. Choose Database.

   The pull-down menu displays three items: Invoices, Accounts, and Customers. These three choices correspond to three databases in Approach.



Memorandum

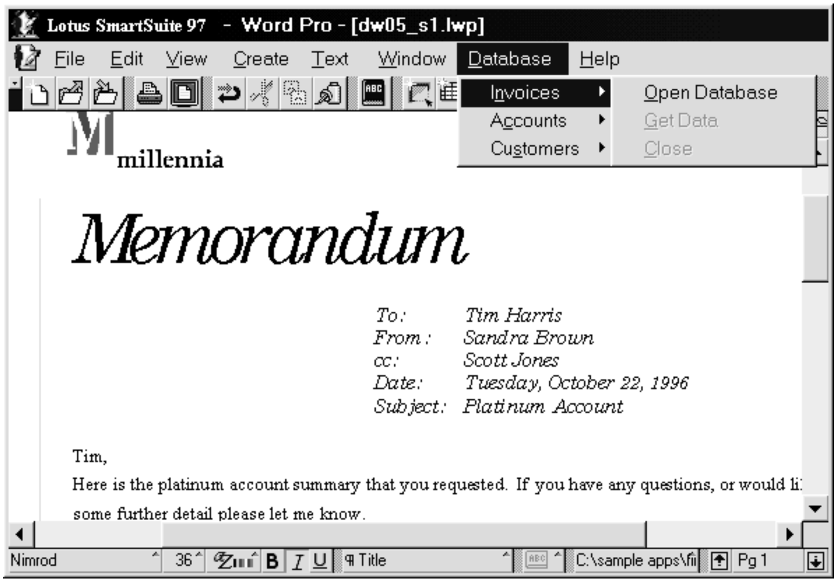
To: Tim Harris
From: Sandra Brown
cc: Scott Jones
Date: Tuesday, October 22, 1996
Subject: Platinum Account

Tim,

Here is the platinum account summary that you requested. If you have any questions, or would li some further detail please let me know.

3. To generate a report about the number of invoices received from a specific vendor, choose Invoices; this item corresponds to the Invoices database in Approach, which contains all the invoice data.

   A cascade displays with three items: Open Database, Get Data, and Close. Only Open Database is active.



Memorandum

To: Tim Harris
From: Sandra Brown
cc: Scott Jones
Date: Tuesday, October 22, 1996
Subject: Platinum Account

Tim,

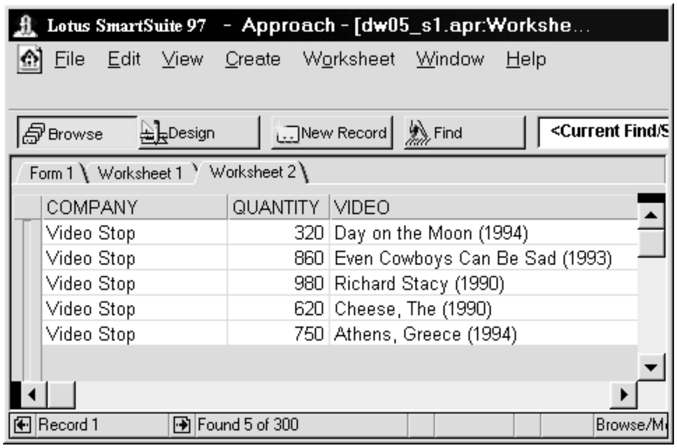
Here is the platinum account summary that you requested. If you have any questions, or would li
some further detail please let me know.

4. Choose Open Database.

   The Approach file DW05_S1.APR opens, via OLE Automation. This file gives you access to the database of sales information.

5. Do a find in Approach to generate the sales data needed for the memo.

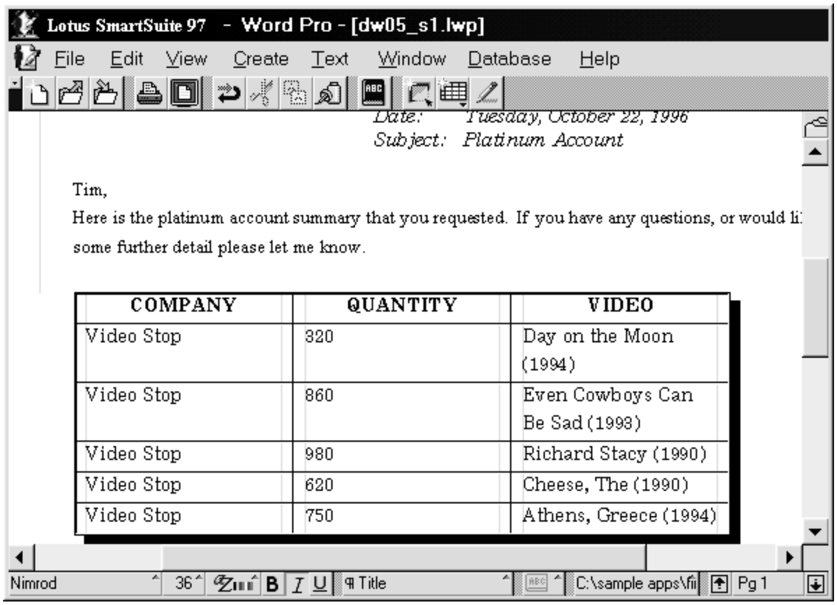
   In this example, the user wants to generate a report that lists all the invoices received from Video Stop. The following illustration shows the found set that results from doing this find.



COMPANY	QUANTITY	VIDEO
Video Stop	320	Day on the Moon (1994)
Video Stop	860	Even Cowboys Can Be Sad (1993)
Video Stop	980	Richard Stacy (1990)
Video Stop	620	Cheese, The (1990)
Video Stop	750	Athens, Greece (1994)

6. Press **ALT+TAB** to return to Word Pro.
7. Choose Database - Invoices - Get Data.

   This creates the table in the Word Pro document and fills it with the data of the found set in the Approach file.



Date: Tuesday, October 22, 1996
Subject: Platinum Account

Tim,

Here is the platinum account summary that you requested. If you have any questions, or would li some further detail please let me know.

COMPANY	QUANTITY	VIDEO
Video Stop	320	Day on the Moon (1994)
Video Stop	860	Even Cowboys Can Be Sad (1993)
Video Stop	980	Richard Stacy (1990)
Video Stop	620	Cheese, The (1990)
Video Stop	750	Athens, Greece (1994)

8. Choose Database - Invoices - Close before exiting Word Pro.

This closes Approach.

The user can now print the report or attach the Word Pro file to a Notes e-mail and send it to the recipient. Upon closing the document, the Database menu item is removed from the menu bar so that the user can open another document without having an unrelated menu item appearing in the menu bar.

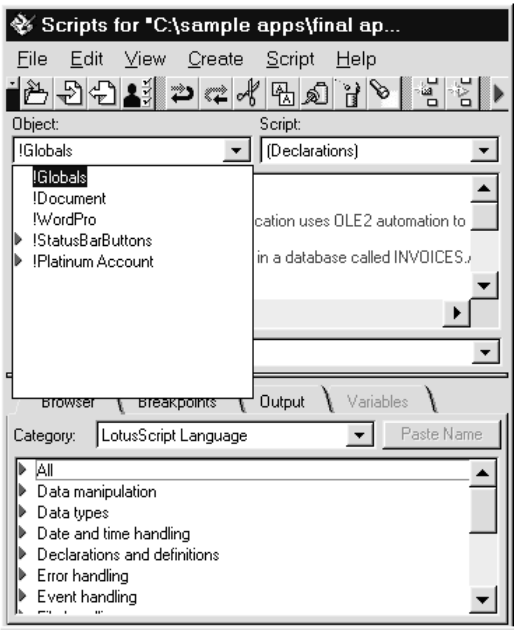
The Video Summary application used to generate this report comprises the functions, subs, and program sections described in the following sections. Lotus strongly recommends that you run the Video Summary application and view the code in the Integrated Development Environment (IDE) as you read the following sections.

The functions and subs are described in the order in which they are typically executed. To view these functions, subs, and program sections in the IDE, open the file DW05_S1.LWP and choose Edit - Script & Macros - Show Script Editor. Select an item in the Object drop-down box and the name of the function, sub, or program section you want to view in the Script drop-down box.

!Globals and !Document

When you open the IDE in DW05_S1.LWP and click the Object drop-down box, two items appear at the top of the list: !Globals and !Document.

Note An exclamation point (!) is a Word Pro LotusScript convention. In 1-2-3, Approach, and Freelance Graphics, Globals appears in parentheses in the Object drop-down box: (Globals).



Object: !Globals

Script: (Declarations)

cation uses OLE2 automation to

in a database called INVOICES.

Category: LotusScript Language

As explained in Chapter 3, !Globals is a default LotusScript object that works similarly to global modules or .BAS files in other BASIC programming environments. Use !Globals or (Globals) to identify declarations and procedures that need to be available to all scripts in your application file.

!Document represents the current document, or an instance of the TextDocument class (in Word Pro, documents are represented by the TextDocument class, not the Document class). In contrast to !Globals, use !Document to identify declarations and procedures that apply only to the scripts of the TextDocument object that represents the current document.

(Declarations) and (Options)

All objects, including !Globals, have two scripts called (Options) and (Declarations). Like other scripts, they are in the IDE in the Script drop-down box. They are worth mentioning here because they are relevant to most of the functions and subs of this application.

In the (Declarations) script, you declare the information that is available to all the subs and functions of the object selected in the Object drop-down box (as opposed to information that is unique to individual subs and functions). Because the subs and functions in !Globals are available to all objects in the application, the information provided in the (Declarations) script of !Globals is available across the entire application. In the Video Summary application, (Declarations) for the !Globals script includes the following information:

- The %Include directive that lets you use the constants in LSCONST.LSS, a file that contains predefined LotusScript constants necessary for the application to process data.
- The definition of DBPATH, a constant that represents the path where the file DW05_S1.APR is located.

  Note If DW05_S1.APR has been moved out of the sample files directory (<*DRIVE*>:\...\SAMPLES\SUITE), make sure that DBPATH matches the path where the file is stored.
- Several variables that are used by different subs and functions in the application.

Every object has an associated (Options) script, which you use to indicate how variables, constants, procedures, and user-defined data types must be declared for that object. Use (Options) to control how data is checked at compile time. In this application, the (Options) section of !Globals has two statements:

```
Option Public
Option Declare
```

Option Public makes all variable, constant, procedure, class, and type declarations that you create in !Globals scripts public by default. In other words, all the variables in any of the functions and subs of !Globals are public, unless they are explicitly declared as private. The advantage of using Option Public (or Option Private) is that you can expose (or hide) declarations to (or from) other parts of the application. So, in this case, Option Public for !Globals means that the declarations are public to the entire application file.

Option Declare indicates that variables must be explicitly declared. In other words, a variable must be declared using the Dim statement, which requires that you specify a data type. The advantage of this is that it makes the code easier to debug because simple errors, such as mistyped variable names, can be caught at compile time.

Functions and subs of the Video Summary application

The following sections describe the functions and subs of the Video Summary application.

Opened

Opened is one of two events of the TextDocument object. This event takes place every time the user opens the file DW05_S1.LWP and is a good place to insert code for initialization tasks. In this example, the Opened sub has been coded with one line that calls the Main sub.

Main

When the document opens, Main runs automatically and calls the following four subs in this order:

- DeleteMenus
- InitAppr
- CreateMenus
- DeleteTable

These subs set up the document and the Word Pro and Approach environment required for the application to work.

DeleteMenus

DeleteMenus deletes Database from the menu bar. It’s unlikely that the menu item is there when the document opens because it is deleted from the menu bar every time the document is closed (DeleteMenus also runs when the document closes). But in the event that the application or the system terminates abnormally, the menu item might still exist in the menu bar. DeleteMenus deletes the item, if it is still there, to ensure that the menu bar doesn’t display Database twice (since Main also runs CreateMenus, which adds Database to the menu bar).

InitAppr

This sub controls the OLE Automation of Approach. It initializes Approach by running the LotusScript CreateObject function, which creates an Approach OLE Automation object. Word Pro acts as the OLE Automation controller, accessing the Approach Application object.

```
Set gApproach = CreateObject("Approach.Application")
```

gApproach is a global variable that is declared in (Declarations). The variable is set to the OLE Automation object ("Approach.Application") that is created by CreateObject. gApproach now contains an OLE Automation object that is an Approach Application object.

The InitAppr sub also performs some error trapping. If the Approach database cannot be found or opened, the On Error GoTo statement points processing to the section in the sub called Problem. This error-handling routine determines what kind of error occurred and, using the MessageBox statement, displays a message.

```
Problem:

   If Err = 208 Then
      MessageBox "InitAppr: Couldn't create Approach" _
         MB_ICONSTOP+MB_OK,"Approach Automation Error. _
           Verify that Approach is installed."
   Else
      MessageBox "InitAppr: Got an error - _
        "+Str$(Err)+":"+Error$, MB_ICONSTOP+MB_OK,"Approach _
         Automation Error.Verify that Approach is installed."
   End If
```

CreateMenus

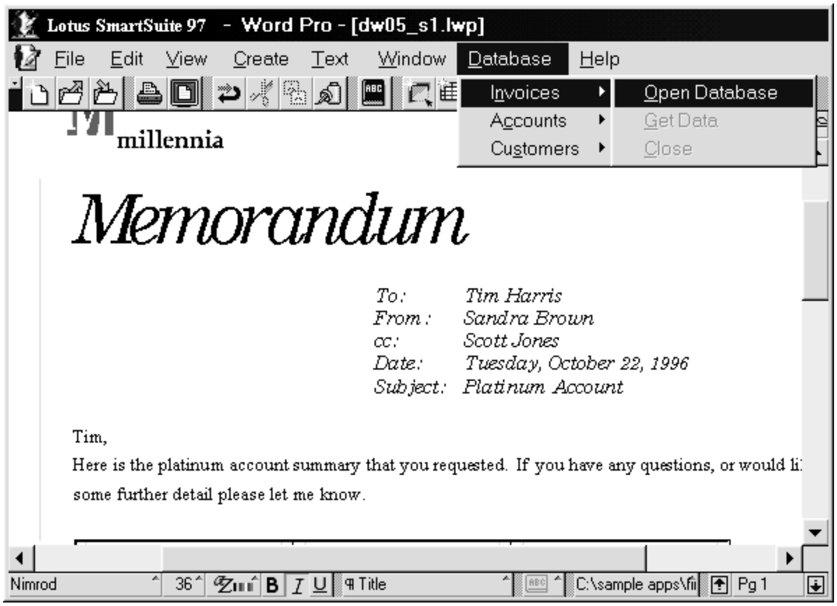
This sub creates the menu item Database and the items in its pull-down menu and cascades. The sub also adds them to the Word Pro menu bar. CreateMenus associates subs with each menu item, and these subs are triggered when the user chooses the menu item.

DeleteTable

This sub deletes the table that was created the last time the application ran. Presumably, the user runs this application every week, so each week a new table is created and stored in the document. The following week, the user runs the application again; this means the previous week's table has to be replaced with the current week's table. If an old table exists in the document, DeleteTable removes it before the new one is created.

Menu_Invoices

This sub runs when the user chooses Database - Invoices - Open Database.



millennia

Memorandum

To: Tim Harris
From: Sandra Brown
cc: Scott Jones
Date: Tuesday, October 22, 1996
Subject: Platinum Account

Tim,

Here is the platinum account summary that you requested. If you have any questions, or would li

some further detail please let me know.

Recall that the CreateMenus sub creates the Database menu and then associates subs with each menu item. When the user selects one of the menu items, the appropriate sub runs. At that point, the Menu_Invoices sub is associated with the Open Database menu item.

Menu_Invoices performs several actions:

- It calls InitAppr, which creates the instance of Approach. Once Approach is open, the application can open the Approach file DW05_S1.APR, using the OpenDocument method of the Approach Application class. Opening the file gives the user access to the database.

  Depending on the directory location of DW05_S1.APR, some text in Menu_Invoices is commented out and some is uncommented. See the note with instructions at the beginning of “OLE Automation using LotusScript with Word Pro and Approach,” earlier in this chapter, for information about what text is commented and uncommented in this sub.

OpenDocument has several arguments:

- The name of the Approach file, DW05_S1.APR.
- The path of the file, represented by DBPATH, a constant explicitly defined in (Declarations), or AutoPATH, a variable declared in Menu_Invoices. AutoPATH is used to store the current path if DW05_S1.APR is stored on the default sample files directory. DBPATH is used if you have moved DW05_S1.APR to another directory.
- The file type. The empty string used here refers to the Approach default file type (.APR).
- The password. The empty string means the file has no password.
- A True/False indicator determining whether the document is to be opened in read-only mode. False means the file is not to be opened as a read-only file, but rather as a read-write file.
- A True/False indicator determining whether the instance of Approach is visible. True means that the instance is visible and that the user can switch to a window displaying the .APR file.

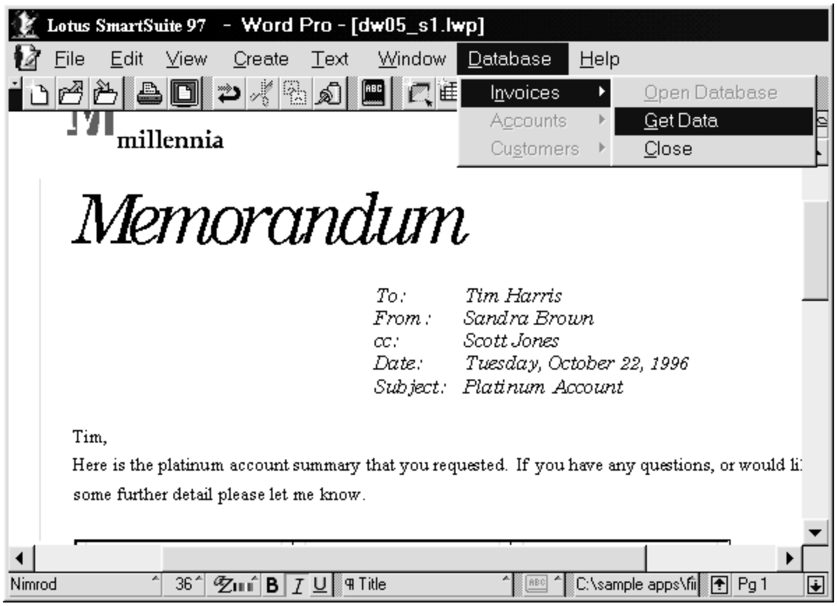
- Although InitAppr creates an Approach OLE Automation object, the object does not automatically appear anywhere on the desktop because Windows does not recognize it as visible. When the user chooses Database - Invoices - Open Database, the Approach OLE Automation object becomes visible as a button in the taskbar. The user can then maximize Approach and do a find for specific records. The following statement in Menu_Invoices makes Approach visible:

  ```
  gApproach.Visible = True
  ```

- When the user first chooses Database in the Word Pro menu bar, the items in the pull-down menu (Invoices, Accounts, and Customers) are all active. The user chooses Invoices and a cascade displays three items: Open Database, Get Data, and Close.

  Open Database is active and Get Data and Close are dimmed (see the previous illustration). This gives the user only one choice: Open Database.

Choosing Open Database not only makes the Approach application visible, it also reverses what is dimmed and what is active: Open Database is dimmed and Get Data and Close become available. The user can now do a find in Approach. After Approach finds the records, the user returns to Word Pro and chooses Database - Invoices - Get Data to create a table displaying the found set in the memo.



Memorandum

To: Tim Harris
From : Sandra Brown
cc: Scott Jones
Date: Tuesday, October 22, 1996
Subject: Platinum Account

Tim,

Here is the platinum account summary that you requested. If you have any questions, or would li some further detail please let me know.

The change in what is dimmed in the Invoices cascade is controlled by a sub called SubMenuToggleVisibility, which is called from Menu_Invoices in the following script. The last argument indicates whether the menu is dimmed or not: True makes it available; False dims it.

```
Call SubMenuToggleVisibility (m111.Items, _
   "&Open Database", False)
Call SubMenuToggleVisibility (m111.Items, _
   "&Get Data", True)
Call SubMenuToggleVisibility (m111.Items, "&Close", True)
Call SubMenuToggleVisibility (gMenuCollection, _
   "A&ccounts", False)
Call SubMenuToggleVisibility (gMenuCollection, _
   "Cu&stomers", False )
```

SubMenuToggleVisiblity

SubMenuToggleVisibility dims menu items. To do this, it runs a sub called ToggleMenu. ToggleMenu steps through the collection of Database menu items: the three items in the pull-down and the nine items composing the three cascades. For each item in the pull-down menu, ToggleMenu calls itself to enable or disable the items in the corresponding menu cascade.

Menu_GetData

Menu_GetData runs when the user chooses Database - Invoices - Get Data. It creates a table in the Word Pro document that contains data extracted from the Approach Invoices database. Therefore, before this sub runs, the user has to do a find in the view Worksheet 2 of the Approach file DW05_S1.APR to specify the data to appear in the memo.

Menu_GetData runs the following functions and subs to extract data from Approach, create the table in Word Pro, and copy the Approach data to the table:

- DeleteTable
- LoadData
- MakeTable
- GetTable
- DisplayTable

To run these functions and subs, Menu_GetData uses the following script:

```
Case gInvoices

   Call DeleteTable
   Call LoadData (gArray(), NumRows, NumCols, gApproach)
   Set CurTable = MakeTable (NumRows+1, NumCols)
   Set CurTable = GetTable
   Call DisplayTable (CurTable, gArray())
```

Menu_GetData is designed to handle the Get Data menu item for the active database, whether it’s Invoices, Customers, or Accounts. Currently, only one database, Invoices, is implemented in the Video Summary Application. If the other two—Accounts and Customers—existed, you could write similar code in Menu_GetData to extract data from those databases and create corresponding tables in the Word Pro document. The Case statements determine which actions are performed depending on which database is active.

The following code in Menu_GetData provides a place to add routines for processing the Customers and Accounts databases:

```
Case gCustomers
'Add code for Customers item in Database pull-down menu.

Case gAccounts
'Add code for Accounts item in Database pull-down menu.
```

LoadData

LoadData, called from Menu_GetData, extracts the data from the Approach Invoices database and creates an array called gArray, in which the data is stored. This array is supplied to DisplayTable as an argument when Menu_GetData runs.

GetColumnNamesInv

To create gArray, LoadData calls a function named GetColumnNamesInv that does the following:

- Returns the number of columns that are currently in the found set displayed in Worksheet 2 of DW05_S1.APR.
- Returns an array containing the field names in the current Approach view.

MakeTable

MakeTable, called from Menu_GetData, is a function that creates the table and places it on the page in the Word Pro document.

GetTable

Get Table, called from Menu_GetData, returns a Table object that MakeTable uses to create the table.

DisplayTable

DisplayTable, called from Menu_GetData, fills the table created by MakeTable with data extracted from Approach and stored by LoadData in the array called gArray.

Menu_CloseDatabase

After the table is created and placed in the Word Pro document, the user should terminate the Approach session. If the user chooses Database - Invoices - Close, Menu_CloseDatabase runs. This closes Approach by removing the OLE Automation object that was originally created when InitAppr ran.

Menu_CloseDatabase also changes the format of the Database menu items. By calling CreateMenus, it refreshes the menu items so that they return to their original state when the user first opened the Word Pro document: Accounts and Customers, items in the Database pull-down menu, are no longer dimmed. Also, Open Database (the first item in the Invoice cascade) is no longer dimmed, and Get Data and Close (the other items in that cascade) are dimmed.

```
Call gApproach.Quit (False)
Call DeleteMenus
Call CreateMenus
```

In addition to returning the menus to their original state, CreateMenus also adds the Database menu and its cascades to the Word Pro menu bar. To avoid having two instances of Database in the menu bar, DeleteMenus is called first to delete the Database menu before CreateMenus adds it again to the menu bar.

PreClose

After closing Approach, the user can close the Word Pro document. Either File - Close or File - Exit closes the document. When the document closes, the PreClose event of !Document runs.

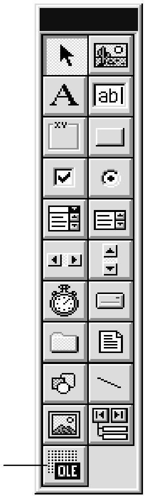
!Document has only two events: Opened and PreClose. PreClose calls DeleteMenus, which removes the Database menu that the Video Summary application adds to the Word Pro menu bar. This is useful if the user closes the document but keeps Word Pro running and then opens a different document that cannot use the Database menu. Deleting the menu when the document closes makes good usability sense and avoids unnecessary errors.

PreClose also closes Approach should the user close the document before choosing Database - Invoices - Close.

```
If Not (gApproach Is Nothing) Then
   Call gApproach.Quit (False)
End If
```

OLE Automation using Visual Basic and 1-2-3

The Map Update application allows a user to use Visual Basic to update a map of the United States created in 1-2-3 to show data such as sales figures for each state. The 1-2-3 file containing the map is embedded in Visual Basic through the OLE container control, a control that lets you embed an OLE object in a Visual Basic form.



OLE container control in Visual Basic

When the Map Update application runs, the object displayed in the OLE container control is updated using OLE Automation.

The Map Update application shows how LotusObjects can be manipulated using Visual Basic. In this simple application, the user must enter numeric data for each state in a text box. In a more elaborate application, the user would not be required to enter data; Visual Basic would most likely extract the data automatically from a database such as Approach, and the map would update as soon as the application ran in Visual Basic. For information about the 1-2-3 Map feature, search on “Maps, overview” in the 1-2-3 Help index. For information about Visual Basic, see the Visual Basic documentation.

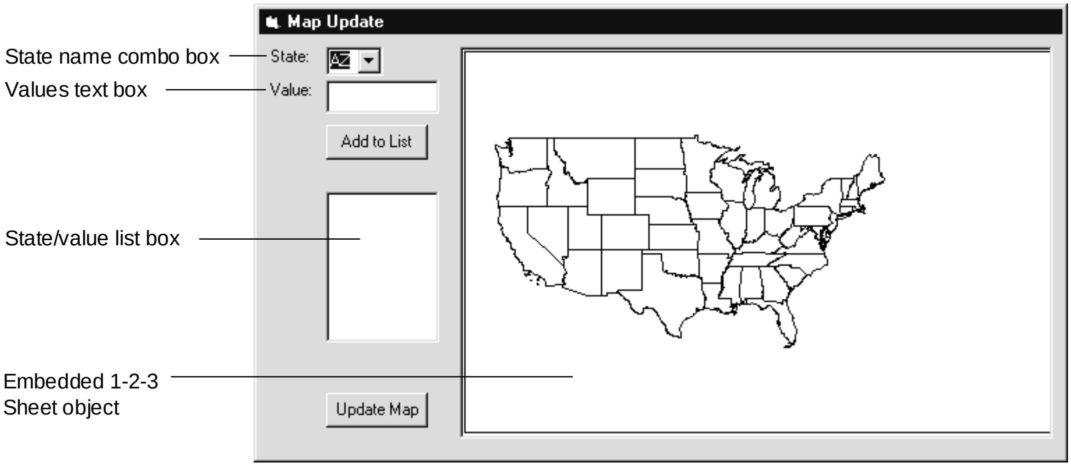
Updating the map

The following steps outline a procedure that the user follows to run the Map Update application, using Visual Basic.

1. Start Visual Basic and open the project DW05_S2.FRM.
2. Select DW05_S2.FRM from the list of forms in the Project window and click View Form.

3. Click Run - Start.

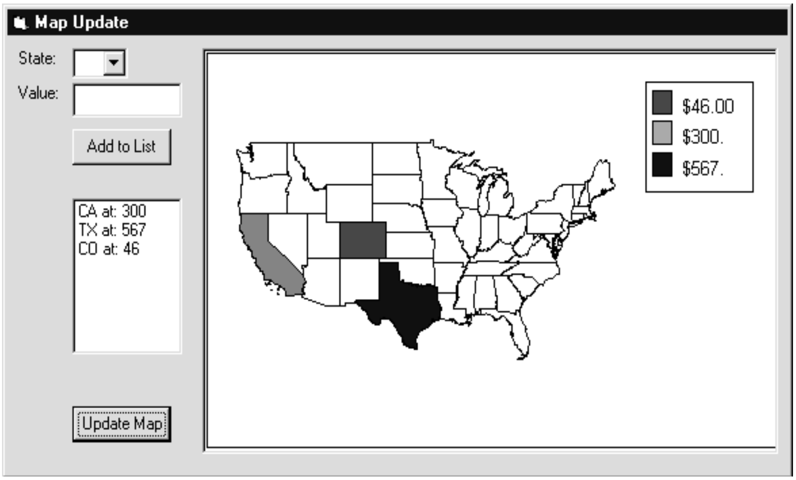
   The following form appears.



4. Select State and select a state name.
5. Enter a numeric value in the Values box.
6. Click Add to List.

   The name of the state and the value appear in the list box.
7. Repeat steps 4 - 6 until you have a list of several states and values in the list box.
8. Click Update Map.

   The map is updated with a legend and color-coding in each of the selected states.



CA at: 300
TX at: 567
CO at: 46

Subs of the Map Update application

The Map Update application consists of three subs described in the order in which they are executed. You can view these subs by opening DW05_S2.FRM and clicking View Code in the Project window.

Form_Load

This sub is the Load event script of the map form that first appears when the user runs the application. Form_Load fills the State box with the list of state names from which the user can select a state.

Add_To_List

This sub is the Click event script for the Add to List button. When the user clicks Add to List, this sub adds the selected state and the numeric value entered in the Values box to the list box.

Update_Map

This sub, the Click event script for the Update Map button, controls the OLE Automation of 1-2-3. After entering the list of states and values, the user clicks Update Map, and the map is updated to display the data entered by the user.

To do this, Update_Map connects the application to the 1-2-3 Sheet object where the map data is stored. The sub writes to the 1-2-3 sheet that generates the map, and 1-2-3 updates the map. Note that no subs or functions are executed in 1-2-3. All programming operations of the application occur in Visual Basic.

The following statements in Update_Map update the 1-2-3 sheet with the map data that the user entered in the Visual Basic Map Update application.

```
OLE1.Object.Ranges.Item("A:F" + CStr(i + 2)).Contents = _
   StateAndSales(i).State
```

```
OLE1.Object.Ranges.Item("A:G" + CStr(i + 2)).Contents = _
   StateAndSales(i).Sales
```

The following code, extracted from the above statements, represents the OLE container control and the OLE Automation object.

```
OLE1.Object
```

OLE1 specifies the Visual Basic OLE container control. Object, which specifies the Object property of the OLE container control, returns the 1-2-3 Sheet object. Object performs the same task as the GetObject function in Visual Basic and LotusScript: it returns an OLE Automation object for the embedded object. In this case the OLE Automation object being returned by the Object property is the 1-2-3 Sheet object.

The text to the right of OLE1.Object represents the data being manipulated in 1-2-3 using OLE Automation.

```
.Ranges.Item("A:F" + CStr(i + 2)).Contents = _
   StateAndSales(i).State
```

```
.Ranges.Item("A:G" + CStr(i + 2)).Contents = _
   StateAndSales(i).Sales
```

Ranges.Item(“A:F” + CStr(i + 2)) and Ranges.Item(“A:G” + CStr(i + 2)) indicate specific cell addresses in the 1-2-3 sheet, in this case cells in columns G and F. The Contents property of the first statement is set to the value entered in the State box. The Contents property in the second statement is set to the value entered in the Value box. These statements are processed for each State / Value pair until the 1-2-3 sheet has been updated with all the data entered in the Visual Basic Map Update application.

Chapter 6
Integration with Notes

Applications built with both SmartSuite and Notes can take advantage of the features in both products. This chapter surveys the basics of integrating the two products using LotusScript. It covers the following subjects:

- Controlling SmartSuite objects from Notes using OLE Automation
- Controlling Notes from SmartSuite using the Notes LotusScript Extension (LSX) module

Note This chapter does not cover accessing Notes databases using Approach. To learn how to access Notes databases using Approach, see Chapter 8.

OLE Automation

Using OLE Automation you can create and manipulate objects of each SmartSuite product from scripts running in Notes. This greatly augments your Notes application development toolbox because it means that you can incorporate SmartSuite product features into your Notes database design.

If you've used OLE Automation to control one SmartSuite product from another, you'll find doing so from Notes to be very easy. If you haven't used OLE Automation before, you may want to read Chapter 5, since it covers in depth the concepts behind scripting with OLE Automation.

Planning ahead

As you design your Notes database, remember that users can access a SmartSuite OLE Automation object through scripts only if they are running Notes Release 4 or later. Also, in order for a user to activate an OLE Automation object, the object's parent SmartSuite product must be installed on the user's machine. Therefore, if an object is embedded in a Notes document and a user hasn't installed the supporting product for the object, the user can't access the object via a script. Although such a limitation may seem obvious, it's worth considering before you deploy a Notes database that only some of the database users can fully run.

Although OLE Automation can add to the power of your Notes database, its performance can sometimes be slow. OLE performance is most affected by the amount of random-access memory (RAM) in a user's machine. Optimize your database design to the performance level of your users' machines. If you know that users won't have a lot of RAM, design your Notes database so that OLE Automation takes place on demand only. For example, if RAM is minimal, don't set a form to activate an OLE object each time a document is opened. Rather, let the users manually activate the object to refresh it whenever they want to see the most current data.

OLE Automation vs. Notes/FX

Notes provides a feature called Notes/FX that lets you exchange data between fields in a Notes document and prepared Notes/FX fields in an embedded object contained by the Notes document. You don't write scripts when you use Notes/FX. Since one of the reasons you use OLE Automation is for exchanging data between Notes documents and SmartSuite objects, it may be unclear when to use each tool.

Use OLE Automation in the following situations:

- The embedded object is to receive data from a large number of Notes fields.
- You want to perform more than just simple data transfers. For example, the embedded object needs to contain computed data based on the fields in the Notes document.
- You want to exchange data between a Notes document and a file in a directory (rather than an embedded object in the Notes document).

Use Notes/FX in the following situations:

- You want the embedded object to include data from a limited number of fields in the Notes document.
- The embedded object is to include real field data, as opposed to computed data based on fields.

Note For more information about Notes/FX, search on "Notes/FX" in your SmartSuite product Help Index.

Simple scripts that use OLE Automation

The following examples are very basic. However, they demonstrate OLE Automation in its simplest forms—namely, creating and working with an embedded object or file, and working with an existing object or file.

Because they're short and academic rather than practical, the code for these basic examples is included in this book, but not on disk. If you want, try the scripts yourself by attaching them to buttons in a Notes document.

Embedding a new 1-2-3 Workbook object using OLE Automation

The following short script for a Notes button does several things. First, it embeds a 1-2-3 Workbook object in the Body field of the current Notes document, and names it Temp. Then, it inserts some text into a range in the 1-2-3 Workbook object and makes the text appear in bold. Finally, it inserts some numbers and tells 1-2-3 to add them.

Note Notes fields must be defined as rich text fields in order to contain embedded objects.

```
Sub Click(Source As Button)
   Dim Workspace As New NotesUIWorkspace
   Dim Uidoc As NotesUIDocument
   Dim Handle As Variant
   Dim MyRange As Variant

   ' Get the current Notes document.
   Set Uidoc = Workspace.CurrentDocument

   ' Put the current Notes document in Edit mode.
   Uidoc.EditMode = True

   ' Move the insertion point to the Body field.
   Uidoc.GoToField("Body")

   ' Create and embed a 1-2-3 Workbook object at the
   ' insertion point.
   Set Handle = _
   Uidoc.CreateObject("Temp","Lotus123.Workbook")

   ' Manipulate the embedded 1-2-3 Workbook object
   ' from Notes:
   Set MyRange = Handle.Ranges.Item("A:A3")
   MyRange.Contents = "Total:"
   MyRange.Font.Bold = True
   Set MyRange = Handle.Ranges.Item("A:B1")
   MyRange.Contents = "3"
   Set MyRange = Handle.Ranges.Item("A:B2")
   MyRange.Contents = "6"
   Set MyRange = Handle.Ranges.Item("A:B1..A:B3")
   MyRange.SmartSum
End Sub
```

Working with a 1-2-3 Workbook object using OLE Automation

You get a handle (a connection) to an OLE Automation object in a Notes document in order to control it from Notes. The previous example uses OLE Automation to create a new 1-2-3 Workbook object named Temp in a Notes document. The following example demonstrates how you can access Temp now that it already exists. Like the previous example, this script is written for a Notes button.

```
Sub Click(Source As Button)
   Dim Workspace As New NotesUIWorkspace
   Dim Uidoc As NotesUIDocument
   Dim Handle As Variant
   Dim MyRange As Variant

   ' Get current Notes document.
   Set Uidoc = Workspace.CurrentDocument

   ' Put current Notes document in Edit mode.
   Uidoc.EditMode = True

   ' Get handle to the embedded 1-2-3 Workbook object
   ' named Temp.
   Set Handle = Uidoc.GetObject ("Temp")

   ' Manipulate the 1-2-3 Workbook object from Notes:
   Set MyRange = Handle.Ranges.Item("A:B1")
   MyRange.Contents = "333"
   Set MyRange = Handle.Ranges.Item("A:B2")
   MyRange.Contents = "666"
End Sub
```

Creating a new 1-2-3 file from Notes using OLE Automation

Instead of working with an embedded 1-2-3 Workbook object in a Notes document (as in the preceding examples), you might want to create a new 1-2-3 file in a directory. The following script for a Notes button creates a new 1-2-3 file named TEMP.123 in the default 1-2-3 directory.

```
Sub Click(Source As Button)
   Dim Handle As Variant
   Dim MyRange As Variant

   ' Create 1-2-3 session.
   Set Handle = CreateObject("Lotus123.Workbook")

   ' Manipulate the 1-2-3 file from Notes:
   Set MyRange = Handle.Ranges.Item("A:A3")
   MyRange.Contents = "Total:"
   MyRange.Font.Bold = True
   Set MyRange =  Handle.Ranges.Item("A:B1")
   MyRange.Contents = "3"
   Set MyRange =  Handle.Ranges.Item("A:B2")
   MyRange.Contents = "6"
   Set MyRange =  Handle.Ranges.Item("A:B1..A:B3")
   MyRange.SmartSum

   ' Save the file as TEMP.123 in the default 1-2-3
   directory.
   Handle.SaveAs "TEMP", Handle.Parent.DefaultPath

   ' Close 1-2-3 session.
   Handle.Parent.Quit
End Sub
```

Writing to a 1-2-3 file from Notes using OLE Automation

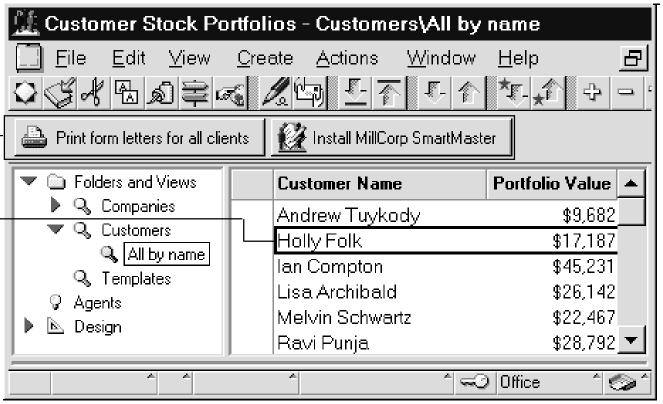
The last example uses OLE Automation to create a new 1-2-3 file named TEMP.123 in a directory. The following script for a Notes button opens the 1-2-3 file TEMP.123, makes some changes to it, saves it, and closes 1-2-3.

```
Sub Click(Source As Button)
   Dim Handle As Variant
   Dim MyRange As Variant

   ' Open the file TEMP.123 in the default 1-2-3 directory.
   Set Handle = GetObject ("TEMP.123", "Lotus123.Workbook")

   ' Manipulate the 1-2-3 file:
   Set MyRange =  Handle.Ranges.Item("A:B1")
   MyRange.Contents = "333"
   Set MyRange =  Handle.Ranges.Item("A:B2")
   MyRange.Contents = "666"
   Set MyRange =  Handle.Ranges.Item("A:B1..A:B3")

   ' Save the file TEMP.123 in the default 1-2-3 directory.
   Handle.SaveAs "TEMP", Handle.Parent.DefaultPath

   ' Close 1-2-3 session.
   Handle.Parent.Quit
End Sub
```

Mid-level scripts that use OLE Automation

The remaining examples in this chapter appear in the Stock Portfolios Notes database. The database file is named DW06_S1.NSF, which you can find in the sample files directory. To use the database and see the examples online, copy the file to your Notes data directory and open it using Notes Release 4 or later.

By default, the database opens in the Customers\All by name view shown here.


Customer Stock Portfolios - Customers\All by name
File Edit View Create Actions Window Help
Print form letters for all clients
Install MillCorp SmartMaster
Folders and Views
Companies
Customers
All by name
Templates
Agents
Design
Office

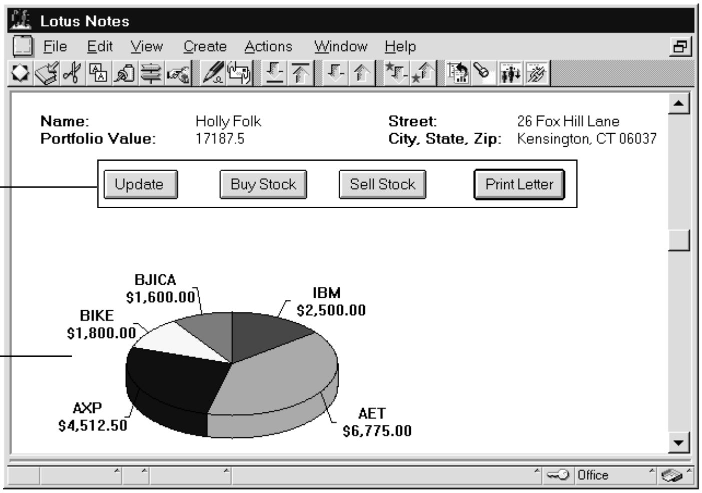
Customer Name	Portfolio Value
Andrew Tuykody	$9,682
Holly Folk	$17,187
Ian Compton	$45,231
Lisa Archibald	$26,142
Melvin Schwartz	$22,467
Ravi Punja	$28,792

Action buttons

Customer document

Each Customer document contains information, such as customer name, address, and portfolio value. Also, when a Customer document is first created, a 1-2-3 Workbook object is embedded in its Body field. This 1-2-3 Workbook object is used to store account transactions.

You can open a Customer document and can click Notes buttons to buy or sell stock. When buying stock, you can choose from among the companies that have Company documents in the Stock Portfolios database. When selling stock, you can choose from among the companies in which the customer owns shares. OLE Automation is used to read and write these transactions to the 1-2-3 Workbook object.



Name: Holly Folk
Portfolio Value: 17187.5
Street: 26 Fox Hill Lane
City, State, Zip: Kensington, CT 06037

Notes buttons

Embedded 1-2-3 workbook

Printing a letter for a specific Customer document

Clicking the Print Letter button in a Customer document sends data from the Customer document to a Word Pro file, tells Word Pro to print, and closes Word Pro. To do this, the script first detaches a Word Pro file named TEMP.LWP to the local drive. Then it uses OLE Automation to write data from the current Customer document to TEMP.LWP, prints the file, and closes Word Pro. After the Word Pro file is printed, TEMP.LWP is deleted from the local drive.

To see the scripts for Print Letter button online, open the Stock Portfolios database, choose View - Design, and examine the Customer form design.

```
Sub Click(Source As Button)
   Dim Session As New NotesSession
   Dim Workspace As New NotesUIWorkspace
   Dim Uidoc As NotesUIDocument
   Dim Doc As NotesDocument
   Dim Handle As Variant

   ' The file, TEMP.LWP, is stored in another document in
   ' the Notes database. The following local sub,
   ' DetachTemplateLetter, is also attached to the Print
   ' Letter button. It locates the document containing
   ' TEMP.LWP and detaches TEMP.LWP to the current drive.
   DetachTemplateLetter

   ' Get the current Notes document.
   Set Uidoc = Workspace.CurrentDocument
   Set Doc = Uidoc.Document

   ' Get object handle to Word Pro, make it visible,
   ' and open TEMP.LWP.
   Set Handle = CreateObject("WordPro.Application")
   Handle.Visible = True
   Handle.OpenDocument Curdrive$() + "\TEMP.LWP"

   ' Get customer name from field CustName in the current
   ' Notes Customer document and write it in two Click Here
   ' Blocks in TEMP.LWP.
   Dim TheCustomer As String
   TheCustomer = Doc.GetFirstItem("CustName").Text
   Call Handle.Foundry.ClickHeres("Name" _
      ).InsertText(TheCustomer)
   Call Handle.Foundry.ClickHeres("Greeting" _
      ).InsertText("Dear " + TheCustomer + ":")
```

```
' Get street address from field CustAddress1 in the
' current Notes Customer document and write to a Click
' Here Block in TEMP.LWP:
Dim Address1 As String
Address1 = Doc.GetFirstItem("CustAddress1").Text
Call Handle.Foundry.ClickHeres("Address" _
   ).InsertText(Address1)

' Get city, state, and zip from field CustAddress2 in the
' current Notes Customer document and write to a Click
' Here Block in TEMP.LWP.
Dim Address2 As String
Address2 = Doc.GetFirstItem("CustAddress2").Text
Call Handle.Foundry.ClickHeres("CityStateZip" _
   ).InsertText(Address2)

' Get the portfolio value from field PortValue in the
' current Notes Customer document and write to a Click
' Here Block in TEMP.LWP.
Dim PortFolioValue As String
PortFolioValue = Doc.GetFirstItem("PortValue").Text
Call Handle.Foundry.ClickHeres("Portfolio Info" _
   ).InsertText("Portfolio value: $" + PortFolioValue)

' Depending on the portfolio value, write one of two
' closing sentences in a Click Here Block in TEMP.LWP.
Dim ClosingSentence As String
If CInt (PortFolioValue) < 5000 Then
   ClosingSentence  = "Feel free to contact us with " _
      + "any questions regarding your account."
Else
   ClosingSentence = "Your account is important to " _
      + "us! One of our brokers will contact you soon."
End If
Call Handle.Foundry.ClickHeres("Closing" _
   ).InsertText(ClosingSentence)

' Write the author's name in a Click Here Block
' in TEMP.LWP.
Call Handle.Foundry.ClickHeres("YourName" _
   ).InsertText(Session.CommonUserName)
   ' Print TEMP.LWP and end the Word Pro session without
   ' saving changes.
   Handle.Print
   Handle.Close False
   Handle.Quit

   ' Delete the file TEMP.LWP from the current drive.
   Kill Curdrive$() + "\TEMP.LWP"
End Sub
```

Printing letters for each account

The script for the Print form letters for all clients action button in the Customers\All by name view in the Stock Portfolios Notes database is very similar to the script for the Print Letter button in a Customer document. However, instead of printing a Word Pro letter for the current Customer document, the action button prints letters for every Customer document in the database. You can examine the scripts attached to the action button in the view design for the view Customers\All by name.

To optimize performance, the Print form letters for all clients action button script only detaches the file TEMP.LWP once. Also, the action button creates a single Word Pro session, but uses it to print every letter.

Large scripts that use OLE Automation

Large scripts use OLE Automation the same way that small ones do. Buying and selling stock in the Stock Portfolios Notes database is accomplished through scripts that are long because they do a lot, not because they use OLE Automation.

Clicking the Buy Stock button in a Customer document in the Stock Portfolios database displays a Notes dialog box in which you can select a stock quote and enter the number of shares to purchase. When you click OK in the dialog box, and then click OK in a confirmation input box, a global sub named WriteToSheet is called. Through this sub, Notes uses OLE Automation to write data to the embedded 1-2-3 Workbook object.

Whenever the sub WriteToSheet writes to the range C:A5 in the embedded 1-2-3 Workbook, the 1-2-3 CellContentsChange event occurs and the handler attached to this event processes the transaction.

To see the scripts for the Buy Stock and Sell Stock buttons, or to see the sub WriteToSheet, open the Stock Portfolios database, choose View - Design, and examine the Customer form design.

To see the scripts for the CellContentsChange event in the embedded 1-2-3 Workbook, create a Customer document, double-click the embedded 1-2-3 Workbook, choose Edit - Scripts & Macros - Show Script Editor, and examine the CellContentsChange event script for range C:A5.

The Notes LSX

The Notes LotusScript Extension (LSX) module enables you to access the Notes object hierarchy from scripts that run in SmartSuite products. When you want to control Notes from scripts running in a SmartSuite product, use the Notes LSX instead of OLE Automation.

Why use the Notes LSX instead of OLE Automation?

- LSXs are type safe. Since the LotusScript compiler can check data types, you can catch and debug type mismatches before run time, which means safer code, created in less time.
- LSXs execute in the same process space as your application scripts. In other words, they are faster than scripts that use OLE Automation because they don't carry the overhead of OLE, and use less virtual memory.

Planning ahead

Since the Notes LSX module wasn't available until Notes Release 4, you'll only be able to use the Notes LSX in a script if you are running Notes Release 4 or later. Your script can connect to a Notes Release 3 database, but both your desktop and server must be running Notes Release 4 or later.

Anyone who uses your application must have database access to a given database in order to connect to it via SmartSuite scripts that use the Notes LSX. Users can't connect to a Notes database that they couldn't otherwise connect to manually.

Note In order to access the Notes LSX module your Notes directory must be in your path.

Loading the Notes LSX

To load the Notes LSX module, enter the following statement in the (Options) section of your SmartSuite product script:

```
UseLSX "*NOTES"
```

Then, when you execute a sub in your script, the Notes object hierarchy is made available. You may find it helpful to press **F2** after entering the statement in (Options). This makes the Notes object hierarchy appear in your Integrated Development Environment (IDE) Browser. When you browse your SmartSuite product classes, you'll find the Notes classes as well.

When the Notes LSX module is loaded, you can access a Notes database from your SmartSuite product using the same syntax that you would use to access one Notes database from another.

For example, the following lines of script open the Notes file PLAN.NSF on the server named Barcelona and then display a message box containing the name of the database:

```
Dim Db As NotesDatabase
Set Db = New NotesDatabase("Barcelona","PLAN.NSF")
MessageBox(Db.Title)
```

Controlling Notes with a Word Pro script

The Stock Portfolios Notes database contains a Word Pro SmartMaster file MILLCORP.MWP. Word Pro documents created using this SmartMaster can access the Stock Portfolios Notes database and read fields from Customer documents.

Setting up the SmartMaster

Click the Install MillCorp SmartMaster action button in the Customers\All by name view of the Stock Portfolios database. The action button detaches MILLCORP.MWP and writes the Notes server name and the Stock Portfolio database file name into two fields in MILLCORP.MWP.

Creating a letter using MILLCORP.MWP

Open Word Pro and create a new file using the SmartMaster MILLCORP.MWP. The Created event occurs. The script attached to this event opens and reads the first Customer document in the Stock Portfolios Notes database and writes the customer name, address, and portfolio value in your new Word Pro file. Before the script finishes executing, it displays an input box asking whether you want to print data for the next Customer document. It repeats this operation until either you indicate that you don’t want to print data for the next Customer document or until it reaches the last Customer document in the Stock Portfolios database.

The following script appears in each document created using the MILLCORP.MWP SmartMaster:

```
UseLSX "*NOTES"  ' Added to (Options)
Sub Created (Source As TextDocument, StyleSheet as String)
   Dim Answer As Integer
   Dim NotesName As String
   Dim NotesAddress As String
   Dim NotesCityStateZip As String
   Dim NotesPortfolioValue As String

   ' When you install MILLCORP.MWP using the action button in
   ' the Customer Stock Portfolios database, the server name
   ' and file name for the Notes database are written in two
   ' document fields in MILLCORP.MWP.
' The alternative to this approach is to specify the
' server name and file name when declaring the
' NotesDB variable.

' Get the name of the Notes database file from Field1 in
' the current Word Pro file.
Dim DbName As String
DbName = .ActiveDocument.DocInfo.FieldManager.Fields( _
   "Field1").Contents

' Get the name of the Notes server from Field2 in
' the current Word Pro file. If the database is on
' your local machine, this will be an empty string.
Dim DbServer As String
DbServer = .ActiveDocument.DocInfo.FieldManager.Fields( _
   "Field2").Contents

' Connect to the Notes database.
Dim NotesDB As New NotesDatabase(DbServer,DbName)

' Get a collection of the documents in the database.
Dim Dc As NotesDocumentCollection
Set Dc = NotesDB.AllDocuments

' Get the first Customer document.
Dim Doc As NotesDocument
For j =  1 To Dc.Count
   Set Doc = Dc.GetNthDocument(j)
   If Doc.HasItem("CustName") Then
      ' Write the Customer name in two bookmark
      ' fields in the current Word Pro file.
      NotesName = Doc.GetFirstItem("CustName").Text
      .GoToBookMark "IncomingName"
      .Type NotesName
      .GoToBookMark "IncomingGreeting"
      .Type "Dear " + NotesName + ":"

      ' Write the Customer street address in a
      ' bookmark field in the current Word Pro file.
      NotesAddress = Doc.GetFirstItem( _
        "CustAddress1").Text
      .GoToBookMark "IncomingStreet"
      .Type NotesAddress
```

```
         ' Write the Customer city, state, and zip
         ' in a bookmark field in the current
         ' Word Pro file.
         NotesCityStateZip = Doc.GetFirstItem( _
            "CustAddress2").Text
         .GoToBookMark "IncomingCityStateZip"
         .Type NotesCityStateZip

         ' Write the Customer portfolio value in a
         ' bookmark field in the current Word Pro file.
         NotesPortfolioValue = Doc.GetFirstItem( _
            "PortValue").Text
         .GoToBookMark "IncomingPortfolioValue"
         .Type "The current value of your portfolio is:" _
            + $" + NotesPortfolioValue + "."

         ' Give user the option to print information from
         ' the next Customer document in the Notes
         ' database.
         If j = Dc.Count Then Exit For
         Answer = MsgBox ("Print Another?", MB_YESNO _
            + MB_IconQuestion, "Continue?")
         If Answer <> IDYES Then Exit For
      End If
   Next
End Sub
```

Chapter 7
Using LotusScript in 1-2-3

Writing scripts in 1-2-3

In 1-2-3, you can use LotusScript to create applications and automate practically any 1-2-3 task. Some of the tasks you can accomplish with LotusScript in 1-2-3 include the following:

- Writing your own @functions
- Creating and displaying your own menus
- Running scripts when a user clicks a button or presses a certain key combination
- Displaying custom dialog boxes you create with the Dialog Editor and running scripts based on the selections users make and the data they enter in the dialog boxes
- Creating your own SmartIcons and attaching scripts to them
- Retrieving data automatically from other applications, such as Notes, or from the Internet
- Integrating 1-2-3 with other products using OLE Automation
- Creating your own 1-2-3 add-ins

Automating tasks without scripts

You can still write macros to automate tasks in 1-2-3. If you are an experienced 1-2-3 macro writer, you might still want to write macros to automate simple tasks. For more information about using macros in 1-2-3, search on “Macros” in the 1-2-3 Help Index.

If you are comfortable writing more complicated macros, you should be able to learn LotusScript relatively quickly. While simple macros are easy to write, scripts are generally more flexible and let you automate far more 1-2-3 functionality than macros do.

Information for upgraders

In previous releases of 1-2-3, you could record your keystrokes and mouse events as macros. Although you can no longer record macros—1-2-3 records all your actions as scripts—you can still write macros yourself. You can also still run the macros you created in previous releases of 1-2-3.

Recording 1-2-3 Classic commands

1-2-3 records any command you execute using 1-2-3 Classic as its corresponding LotusScript statement. For example, suppose you use 1-2-3 Classic to format the currently selected range as Currency with two decimal places. 1-2-3 records the 1-2-3 Classic keystrokes

```
/RFC2
```

as

```
Selection.Format "US Dollar",2
```

If a 1-2-3 Classic command has no corresponding LotusScript statement, 1-2-3 does not record the command. A 1-2-3 Classic command that has no corresponding command on the 1-2-3 main menu also has no corresponding LotusScript statement. For more information about using the 1-2-3 Classic in this release, search on “1-2-3 Classic” in the 1-2-3 Help Index.

Running a macro from a script

You can easily incorporate an existing macro into a new script you write by calling the macro from the script. To call a macro, use the MacroRun method. For example, to call a macro named ROLLUP, use:

```
[Rollup].MacroRun
```

For more information about running macros from scripts, search on “MacroRun method” or “MacroRunText method” in the 1-2-3 Help Index.

Upgrading macro buttons

When you open a .WK4 file, 1-2-3 automatically converts macro buttons that you created with previous releases of 1-2-3 into buttons that run scripts.

If a button ran a macro stored in a sheet, 1-2-3 creates the following Click event script for the button:

```
Sub Click(Source As ButtonControl)
    [file].Ranges("range")MacroRun
End Sub
```

file is an optional file reference. You need to specify the file only if the macro is stored in a different file than the button, for example, if the macro is stored in a macro library.

range is the name or address of the first cell in the macro.

For example, the following sub runs the macro named INTEREST stored in the active file D:\LOTUS\WORK\MYMACROS.WK4:

```
Sub Click()
    [d:\lotus\work\mymacros.wk4].Ranges("interest").MacroRun
End Sub
```

If a button ran a macro stored with the button in the Assign Button dialog box, 1-2-3 creates the following Click event script for the button:

```
Sub Click(Source As ButtonControl)
    .MacroRunText("{macro_command}")
End Sub
```

For example, the following sub runs a macro that enters the label “Weekly Status Report” in the current cell and changes the font and column width:

```
Sub Click(Source As ButtonControl)
    .MacroRunText |{CELL-ENTER "Weekly Status Report"} _
    {STYLE-FONT "Baskerville"}{COLUMN-WIDTH 30}|
End Sub
```

Note If you open a .WK4 file that contains macros stored with buttons and then subsequently save the file as a .WK4 file, macros stored with buttons are preserved.

Upgrading custom dialog boxes

In previous releases of 1-2-3, after you created a dialog box in the Dialog Editor, you copied it to the Clipboard and pasted it in a worksheet file so that you could use it in a macro. While you can continue to run the macros that display these dialog boxes, you cannot upgrade them. For information about creating new dialog boxes with the Dialog Editor, see Chapter 3.

The 1-2-3 object model

Before you begin writing scripts in 1-2-3, you should take some time to understand the 1-2-3 object model, which describes all the 1-2-3 objects and their organization.

Before continuing to read this section, you should understand the concepts of classes and objects, containment and inheritance, and dot notation. For this information, see Chapter 2.

Like all LotusObjects, objects in 1-2-3 have properties, methods, and events. Properties are characteristics that describe the state of an object. Methods are subs or functions that can be performed by the object. Events represent external actions performed on objects by users, the application, or the operating system.

The following sections describe important 1-2-3 objects and their relationships to one another:

- 1-2-3 containment hierarchy
- 1-2-3 inheritance relationships
- Predefined global product variables that make it easier for you to specify commonly used 1-2-3 objects
- Collection classes in 1-2-3
- Examples of how to specify 1-2-3 objects in scripts

1-2-3 containment hierarchy

Like all SmartSuite product object models, the 1-2-3 object model is organized by containment hierarchies, which describe the containment relationships of the 1-2-3 classes. The 1-2-3 containment hierarchy is constructed logically along the lines of what you might think of as “containment” in the 1-2-3 user interface. For example, just as workbooks contain ranges and sheets, the Ranges and Sheets objects are contained by the Document object. The 1-2-3 containment hierarchy begins with the outermost 1-2-3 object, the Application class, and works in through workbook files (.123), sheets, ranges, charts, OLE objects, and graphic objects. Contained objects can be identified by using a property of the container. Look for a property of the container in one of the following configurations:

- The property is included in the definition of the container class.

  In the most simple case, you can tell the containment relationship from the class description. The property name is part of the class definition. You can find the property listed in the Browser, and by selecting the property and pressing **F1**, you can see a description of the property in Help.

For example, the Sheets class is contained by the Document class through the Sheets property.

- The property identifies a collection of the contained objects.

  This case also allows you to tell the containment relationship by looking at the class description. You must also know the relationship between the object you want to identify and the collection. Identify the object by referring to an element of the collection.

  For example, all the ranges for a workbook (Document object) are in a collection of Range objects. The collection is a Ranges object and is contained in the Ranges property of the Document object. Each Range object is an element of the Ranges object.

  You can manipulate a collection object just as you would any other object. For example, the following script creates the object r, which represents the collection of ranges in the current workbook:

  ```
  Dim r as Ranges
  Set r = CurrentDocument.Ranges
  ```

  Because collections are special classes, however, you need to use LotusScript to access individual elements in a collection. For example, you can use the following statement to access the second range in a collection:

  ```
  CurrentDocument.Ranges(1)
  ```

  LotusScript uses indexes to identify the position of an element in a collection. Because the first index is always 0, the number 1 indicates the second range in the collection. Note that 1-2-3 does not guarantee the order or elements in a collection.

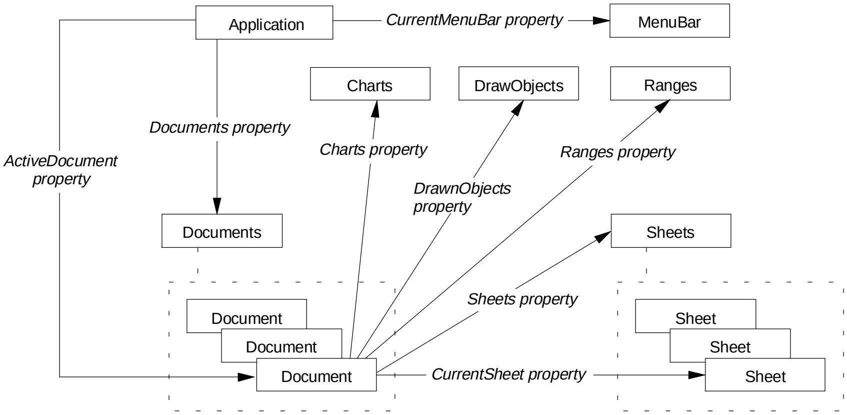
  In addition to using index numbers, you can also access individual items in a collection by using their names. For example, you can use the following statement to access the range named Sales in the current document:

  ```
  CurrentDocument.Ranges("Sales")
  ```

  For more information about collections, see “Collection classes in 1-2-3” later in this chapter.

Important containment relationships in 1-2-3

The following diagram illustrates the containment relationships of the most important 1-2-3 classes.



The most important relationships in the 1-2-3 containment hierarchy are described next.

Application class as container

The Application class in 1-2-3 represents a 1-2-3 session started from the executable 123W.EXE. The Application class contains three important 1-2-3 classes: ApplicationWindow, Documents, and MenuBar.

To refer to a document in the current 1-2-3 session, specify its position in the Documents collection. For example, the following statement refers to the first document in the current 1-2-3 session:

```
CurrentApplication.Documents(0)
```

For information about using the MenuBar class to customize 1-2-3 menus, see “Creating a custom menu” later in this chapter.

Document class as container

The Document class in 1-2-3 represents an active workbook (.123 file). The Document class contains other important 1-2-3 classes: Sheets, OLEObjects, Charts, DrawObjects, and Ranges.

To refer to a sheet in the current workbook, you can specify either the position of the sheet in the Sheets collection or the name of the sheet. For example, the following statement displays the second sheet in the current workbook:

```
CurrentDocument.Sheets(1).GoTo
```

The following statement displays the sheet named Sales:

```
CurrentDocument.Sheets("Sales").GoTo
```

Objects that represent ranges, charts, drawn objects, and OLE objects are also contained by Document objects. Using the containment relationships discussed earlier in this chapter, you can identify individual objects in either of the following ways:

- Using the object’s name

  ```
  ' A chart named MyChart in the current workbook
  CurrentDocument.Charts("MyChart")

  ' A range named Q2Sales in the current workbook
  CurrentDocument.Ranges("Q2Sales")

  ' A rectangle named Rectangle 1 in the current workbook
  CurrentDocument.DrawObjects("Rectangle 1")
  ```

- Using indexes to identify the position of an object in a collection

  ```
  ' The third OLE object in a collection.
  CurrentDocument.OLEObjects(2)
  ```

Identifying containers using the Parent property

The Parent property plays an important, though not apparent, role in the 1-2-3 containment hierarchy. Most 1-2-3 objects have a Parent property, which returns the parent (or container) object of a specified object. Sometimes when you write scripts in 1-2-3, it is useful to be able to find a parent, or container, object through one of its children, or contained, objects. For example, you might already have a Range object, but might want to know what Document object contains it.

The following example updates a range, and then saves the workbook that contains the range:

```
[MyRange].RangeFill
MyRange.Parent.Save
```

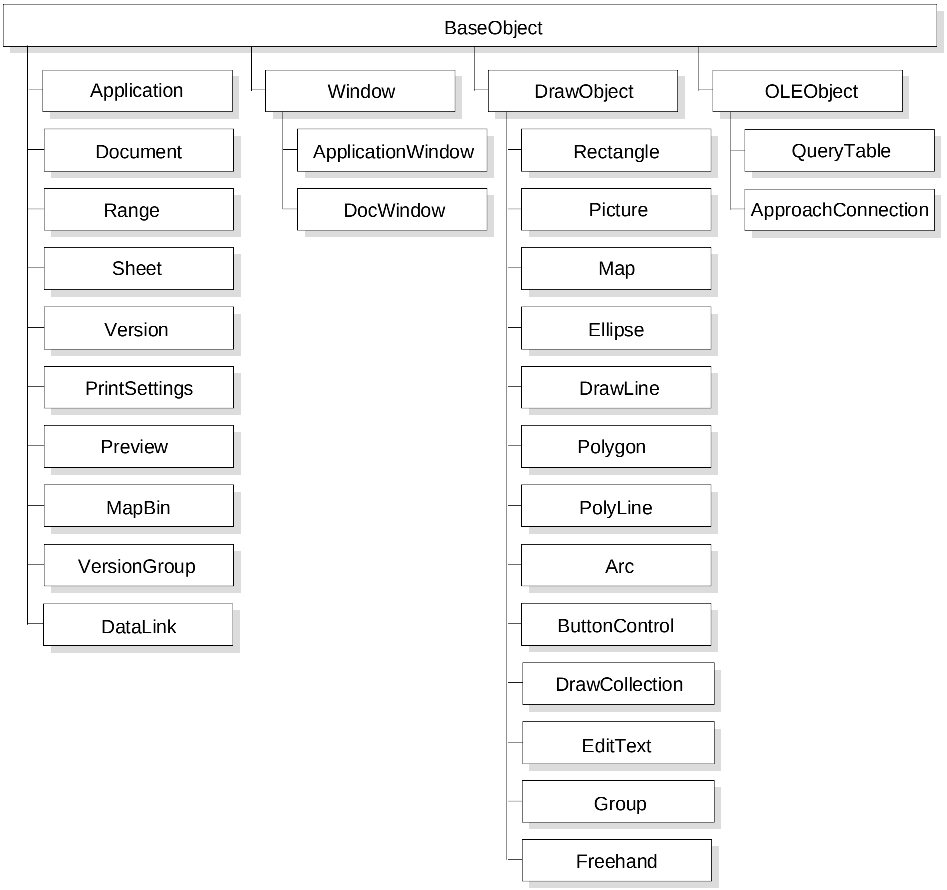
The Parent property is especially useful when you are working with events. An event receives the object that owns the event as the parameter Source. Source identifies the object for which an event occurs. The Parent property lets you access the source object’s container object.

For example, the following code, which is executed when the user clicks a button, uses the Parent property to identify the file that the button is in and then closes that file:

```
Sub Click(Source As ButtonControl)

   ' The sheet that contains the button is its parent

   Set X = Source.Parent

   ' The file that contains the sheet is its parent

   X.Parent.Close

End Sub
```

1-2-3 inheritance relationships

The following diagram shows the the important inheritance relationships in 1-2-3 classes. For more information about inheritance, see “Inheritance” in Chapter 2.



1-2-3 abstract classes

1-2-3 has the following abstract classes, which exist only to derive other classes, called concrete classes. You cannot create an instance of an abstract class.

- The BaseCollection class is the base class for all collection classes in 1-2-3. For more information about collections and a diagram of the important 1-2-3 classes derived from BaseCollection, see “Collection classes in 1-2-3,” later in this chapter.
- The BaseObject class is the base class for all 1-2-3 classes. All 1-2-3 classes inherit the following properties from BaseObject: Application, Description, IsValid, Name, Parent, and VersionID.
- The Window class is the base class for the 1-2-3 window classes: ApplicationWindow and DocWindow.
- The DrawObject class is the base class for all 1-2-3 drawn object classes. The following 1-2-3 classes inherit from the DrawObject class: Arc, ButtonControl, DrawCollection, DrawLine, EditText, Ellipse, Freehand, Group, Map, Picture, Polygon, Polyline, and Rectangle.

1-2-3 predefined global product variables

1-2-3 supports several predefined global product variables that greatly simplify specifying objects.

Variable	*Description*
CurrentApplication	Represents the current session of 1-2-3. Uses the properties and methods of the Application class.
CurrentDocument	Represents the current 1-2-3 workbook (.123 file) in the current 1-2-3 session. Uses the properties and methods of the Document class.
CurrentWindow	Represents the window in which 1-2-3 displays the current document. Uses the properties and methods of the DocWindow class. If no files are open, then CurrentWindow represents the 1-2-3 application window and uses the properties and methods of the ApplicationWindow class.
Selection	Represents the currently selected object, for example, the currently selected range, chart, or graphic object.
ThisDocument	Represents the document that contains the script that is currently running. Uses the properties and methods of the Document class.

Global product variables replace all the objects that are containers for the current object. For example, to use the Name property to determine the name of the current document, use the following syntax:

Correct:

```
n = CurrentDocument.Name
```

Do not use other variables to fill in the containment tree.

Incorrect:

```
n = CurrentApplication.CurrentDocument.Name
```

1-2-3 collection classes

A collection is a special class that consists of a set of objects of a particular type, grouped together to form a separate object. For example, the Sheets class contains all the sheets in a particular workbook. Each object in a collection is called an element in that collection. You can access elements in a collection in two ways:

- Iteration is the process of stepping through a collection and acting on each element in the collection. Use the LotusScript ForAll statement to iterate through a collection.

  For example, the following statement makes all the charts in the current workbook pie charts:

  ```
  ForAll x in CurrentDocument.Charts
    x.Type = 6
  End ForAll
  ```

  Note that you can only access named ranges by iterating through the Ranges collection. For example, the following statement lists all the named ranges in the current workbook:

  ```
  ForAll x in CurrentDocument.Ranges
    Print x.Name
  End ForAll
  ```

- Indexing is the process of using the Item method or the indexing syntax to access a specific object in the collection.

  The following example lists the names of the sheets in the current workbook:

  ```
  Sub ListSheets
     Dim n As Long
     n = CurrentDocument.Sheets.Count
     For x = 0 To n - 1
        Print CurrentDocument.Sheets.Item(x).Name
     Next
  End Sub
  ```

Note that you can access any Range object by using the string equivalent of its address as the index into the Ranges collection. For example, the following code changes the background color of the range A:A1..A:B10 to red:

```
CurrentDocument.Ranges("A:A1..A:B10" _
   ).Background.BackColor.ColorName = "red"
```

Methods and properties for collection classes

Like other classes, collection classes have their own properties and methods. You can use these properties and methods to control individual elements in the collection or to control the entire collection.

Each 1-2-3 collection class has the following methods:

- Item returns the specified element in the collection.
- Next returns the next element in the collection.
- Open resets the Next method so that it returns the first element of the collection.

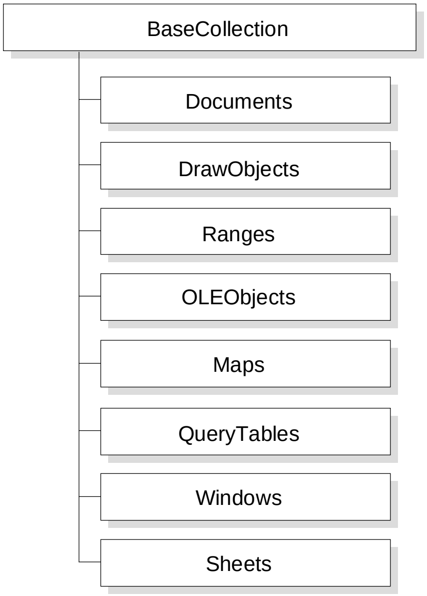
Each 1-2-3 collection class has the following property:

- Count returns the number of elements in the specified collection.

For more information about each of these methods and properties, search on “Methods (LotusScript)” or “Properties (LotusScript)” in the 1-2-3 Help Index.

Common collections

The 1-2-3 object model contains a number of collection classes. You will use some of these classes more than others. The following diagram shows the most commonly used collection classes in 1-2-3. For more information about any of the 1-2-3 collections, search on “Classes (LotusScript)” in the 1-2-3 Help Index.



Identifying objects in 1-2-3

When you write scripts in 1-2-3, you can identify a 1-2-3 object with its name, provided you enclose the name in square brackets ([ ]). The following examples show several 1-2-3 objects identified by name:

```
' Bold data in the range February.
[February].Font.Bold = True
```

```
' Change the chart MyChart to a doughnut chart.
[MyChart].Type = $Doughnut
```

```
' Protect all cells in the sheet Budget.
[Budget].IsProtected = True
```

You can also identify a range with its address enclosed in square brackets.

```
' Add a border and border style to A:A1..A:B10.
[A:A1..A:B10].OutlineBorder.Style = $SolidBorder
```

```
' Change the background color of sheet A to cornflower.
[A].Background.BackColor.ColorName = "Cornflower"
```

By default, names within square brackets identify objects in the workbook that has the focus. To identify objects in other workbooks, use a file reference with the following format:

```
[<<FileName>>ObjectName]
```

For example, the following statement identifies a range named MyRange in the workbook D:\LOTUS\WORK\123\MYFILE:

```
[<<D:\Lotus\Work\123\MyFile>>MyRange].CopyToClipboard
```

Type qualifiers

If multiple objects in a workbook share the same name, use a type qualifier to specify which object you want the script to act on. The syntax for an object specification that includes a type qualifier is as follows:

```
[ObjectName:Type]
```

For example, suppose a workbook contains a map named Sales and a chart named Sales. Use the following reference for the map:

```
[Sales:Map]
```

Use the following reference for the chart:

```
[Sales:Chart]
```

If you do not include a type qualifier in an object specification, 1-2-3 first looks for a range with the specified name, because ranges have the highest order of precedence in a workbook. The order of precedence for all other objects within a workbook is neither guaranteed nor consistent.

For example, if a workbook contains a range named Sales, a chart named Sales, and a map named Sales, the following statement always selects the range:

```
[Sales].Select
```

If a workbook contains only a chart named Sales and a map named Sales, unexpected results occur. The statement selects either the map or the chart, arbitrarily. To avoid this, you should not assign the same name to two objects in the same workbook. However, if a workbook contains multiple objects with the same name, always include type qualifiers in your object specifications. For more information about naming objects in 1-2-3, search on “Naming, conventions for” in the 1-2-3 Help Index.

Leading dot notation

Leading dot notation can save you time when you are writing a script. For example, the following statement invokes the Save method of an object.

```
Call .Save()
```

The following statement also invokes the Save method of an object, but involves more typing.

```
MyDocument.Save()
```

1-2-3 resolves the leading dot to the nearest object in the script that the method can act on. For example, because the Save method applies to a Document object, but not to a Chart object, if no object is selected, the Save method in the following script would not try to save the object MyChart. It would save the object MyDocument:

```
MyDocument.Open
Select.MyChart
Selection.Type = $Bar
.Save()
```

If an object is selected, either with the Select method or via the user interface, the leading dot resolves to the currently selected object. For example, the following code selects a range and then formats the currently selected range:

```
[A:A1..A:D10].Select
.Font.FontName = "Baskerville"
.Font.Size = 10
.Font.Italics = True
```

Recording scripts in 1-2-3

1-2-3 lets you record actions you perform in 1-2-3 as scripts.

To record a script in 1-2-3:

1. Choose Edit - Scripts & Macros - Record Script.
2. Type a name for the script in the “Script name” box.

   You must enter a name for your script. 1-2-3 does not provide a default name, and displays an error if you do not name your script.
3. Select a workbook in which to store the recorded script. You can store the script in any active workbook. 1-2-3 stores the recorded script in the (Globals) object in that file.
4. Click Record.
5. Perform the task you want to record.
6. To stop recording and display the recorded script in the Script Editor, choose Edit - Scripts & Macros - Stop Recording.

Recording into an existing script

You can record actions you perform in 1-2-3 into a script that you previously wrote or recorded.

To record into an existing script:

1. Choose Edit - Scripts & Macros - Show Script Editor.
2. Select the object and then the script you want to add to.
3. In the Script Editor, click at the end of the line after which you want to begin recording and press **ENTER**.

   Note Always begin recording at the beginning of a new line, not in the middle of an existing line.
4. In the Integrated Development Environment (IDE), choose Script - Record at Cursor.

   1-2-3 displays recording controls.
5. Minimize the IDE so that you can see more of the 1-2-3 window.
6. Perform the task you want to record.
7. To stop recording and display the recorded script in the IDE, choose Edit - Scripts & Macros - Stop Recording.

Using the IDE in 1-2-3

To open the IDE in 1-2-3, choose Edit - Scripts & Macros - Show Script Editor. The IDE displays the script that was last displayed or recorded, or a blank default script for the currently selected object if no scripts have yet been written or recorded.

You can also open the IDE by selecting an object in 1-2-3 and then choosing Show Script Editor from the shortcut menu. The IDE appears and displays the first script associated with the object. If the object has no scripts associated with it, the IDE displays an empty script.

The Object drop-down box in the IDE lists the 1-2-3 objects. When you first open the list, you see the following items:

- (Globals), which lists the global subs and functions.
- The name of the Document object (.123 file) associated with this IDE window. This object is always the one that represents the current workbook.
- Lotus123, which is the name of the Application object.
- The name of the currently selected object. For example, if the cell pointer is in cell B:A25 when you open the IDE, B:A25 appears in the Object list.
- The names of all objects in the workbook that have scripts attached.

The Script drop-down box lists available scripts for the object selected in the Object drop-down box. Although the scripts available for an object differ according to the scope and function of the object in 1-2-3, the following scripts are available for most objects:

- (Options) scripts contain statements that specify LotusScript language options, external .LSO or .LSS files, and some constants used by external files.
- (Declarations) scripts contain declaration statements, constant definitions, and class definitions.
- Initialize scripts set up variables declared in a (Declarations) script.
- Terminate scripts clean up variables declared in a (Declarations) script.
- Event procedures for an object define how that object should respond to particular events that it receives, such as being clicked, moved, or opened.

Writing scripts for objects that do not appear in the Object list

Some objects you want to write scripts for cannot be selected in 1-2-3 and do not appear in the Object drop-down box. When you write scripts for these objects, you must manually create an object variable.

For example, suppose you write a script that you want to run when a particular DocWindow object gets the focus. You cannot select a DocWindow object in 1-2-3 or from the Object drop-down box. To associate the event script named MyGetFocus with the event GetFocus of the first DocWindow object, you could use the following script in the Opened event script:

```
Sub Opened(Source As Document)
    Dim DocWindow1 as DocWindow
    Set DocWindow1 = ThisDocument.DocWindows(0)
    On Event GetFocus From DocWindow1 Call MyGetFocus
End Sub
```

You could then put your GetFocus event script in the (Globals) object in the IDE.

Using the Dialog Editor in 1-2-3

To open the Dialog Editor in 1-2-3, choose Edit - Scripts & Macros - Show Dialog Editor. 1-2-3 saves dialog boxes you create with the Dialog Editor with the .123 file in which you created them. For more information about using the Dialog Editor, see Chapter 3.

Customizing the 1-2-3 user interface

You can use LotusScript to customize practically every aspect of the 1-2-3 user interface. This section describes the steps necessary to do the following:

- Create a new menu item and attach a script to it
- Customize an icon and attach a script to it
- Create a button and attach a script to it

To see examples of scripts attached to custom menus, SmartIcons, and buttons, including sample code, see “Top Tasks” later in this chapter.

Attaching a script to the Actions menu

You can make it easy for users to run your scripts by displaying menu choices for the scripts on the Actions menu. To run a particular script, the user simply chooses the corresponding menu choice from the Actions menu. Note that the Actions menu is only displayed in the 1-2-3 menu bar when there are scripts attached to it.

To attach a script to the Actions menu:

1. Choose Edit - Scripts & Macros - Global Script Options.
2. Select the name of an open workbook from the "Edit options for script from" drop-down box.
3. Select the name of a script from the "Scripts" box.
4. Click Edit Options.
5. Enter the text you want to appear in the Actions menu in the "Menu command on Actions menu" box.

   When you enter command names, an ampersand (&) followed by a character creates a keyboard shortcut for a command. The letter that follows the ampersand appears underlined; the user can choose this command from the keyboard by pressing **ALT** plus the underlined letter. For example, if you enter First &Quarter, 1-2-3 displays First Quarter in the Actions menu. The user can press **ALT+Q** to select the command. To display an ampersand in the command name, enter two ampersands (&&). For example, to display B&W, enter B&&W.
6. (Optional) Enter a description of the command in the "Help text for menu command" box.
7. Click OK to return to the Global Script Options dialog box.
8. Click Done.

Tips for attaching scripts to the Actions menu

Keep the following in mind when you attach scripts to the Actions menu:

- You can attach only subs that have no parameters to the Actions menu.
- If the script is stored in a workbook, it appears on the Actions menu only when the workbook is in memory.
- If you delete the script in the Script Editor, 1-2-3 automatically removes the entry from the Actions menu and deletes the shortcut key assigned to it.

- If you make any changes to the script in the Script Editor, 1-2-3 checks to see if it is still valid for attaching to the Actions menu. If the script is invalid, 1-2-3 automatically removes the menu command to which it is attached from the Actions menu.

  For example, if you add parameters to a global sub that appears on the Actions menu, 1-2-3 automatically removes the sub from the menu.

Attaching a script to an icon

SmartIcons provide a quick, simple way to do many 1-2-3 tasks. When you attach a script to an icon, it runs when the user clicks the icon. You can attach a script to an existing icon in 1-2-3 or to an icon that you design yourself. For example, you can create a script that enters a company name and address in a special style in a sheet. Then you can create an icon with the company logo on it and assign the script to this icon.

Designing a new icon

To create a custom icon, you must start with a new, blank icon and add a bitmap to it.

To create the bitmap for your icon, you can do one of the following:

- Copy the bitmap of an existing icon and edit it
- Start with a blank icon and paint your own bitmap
- Copy a bitmap to the Clipboard from another application, such as Paintbrush, and paste it into a blank icon

To design a new icon:

1. Choose File - User Setup - SmartIcons Setup.
2. Click Edit Icon.
3. Click Create a New Blank Icon to create a new icon or select an icon to modify from the “Available icons you can edit or copy” box.
4. Edit the icon by applying or changing colors.
5. Enter the icon bubble help in the “Description” box.
6. Click Save As to name and save the new bitmap.
7. Click Done to return to the SmartIcons Setup dialog box.

   The new icon appears in the “Available icons” box.
8. Click OK.

Attaching a script to an icon

You can attach a script to an existing icon or to a custom-designed icon.

To attach a script to an icon:

1. Choose File - User Setup - SmartIcons Setup.
2. Click Edit Icon.
3. Click an icon in the “Available icons” box.
4. Click Attach Script.
5. Select the name of an open workbook from the “From” drop-down box.
6. Select the name of a script from the “Script name” box.
7. Click Attach.

Attaching a script to a button

You can create a button on a sheet and attach a script to the button. When the user clicks the button, the script runs.

To create a button:

1. Choose Create - Button.
2. Position the mouse pointer in the sheet where you want the button to appear.
3. Do one of the following:
   - To create a button in the default size, click the sheet.
   - To size the button, drag across the sheet and release the mouse button when the button is the size you want.

     When you release the mouse button, 1-2-3 opens the Script Editor, which contains an empty sub for the Click event for the button. The statements you enter in the sub will execute when the user clicks the button.

Tip When you create a button, 1-2-3 assigns it a default name and button text. To change the name of the button, button text, or other properties, right-click the button and choose Button Properties from the shortcut menu.

Attaching a script to a picture

Although you cannot display a picture on a button you create in 1-2-3 with the Create - Button command, you can add a picture, such as a bitmap, to the sheet and then attach a script to the picture. The script runs when the user selects the picture.

Adding a picture to a sheet

To add a picture to a sheet:

1. Choose Create - Drawing - Picture.
2. Select the drive and folder containing the picture file from the “Look in” box.
3. (Optional) Select the file type of the picture from the “Files of type” drop-down box.
4. Select the picture file you want from the list.
5. Click Open.
6. Click the sheet at the point where you want to place the top left corner of the picture.

Tip You can also bring a picture into 1-2-3 by copying it to the Clipboard and pasting it in the sheet.

Attaching a script to a picture

To attach a script to a picture:

1. Right-click the picture.
2. Choose Show Script Editor from the shortcut menu.
3. Choose the Selected event script from the “Script” drop-down box.
4. Write and save the script.

Now, when the user selects the picture, the script runs.

Team computing in 1-2-3

Team computing features help you communicate, collaborate, and coordinate with others in your organization to work together more effectively. You can use LotusScript to automate the following Team Computing features in 1-2-3:

- Sending a mail message
- Sending or routing a range or workbook

Sending a mail message with an attachment

TeamMail provides enhanced electronic mail support that lets you distribute an entire 1-2-3 workbook or a broadcast message to co-workers. TeamMail is useful when you only want to distribute information without receiving input from those who receive the information.

This example assumes the user created a sheet that is updated daily with stock-price information. It sends an electronic mail message that includes a bitmap image of the sheet to a specified group of people. This example is written to be run from a button at the top of the sheet.

```
Sub MailDaily
   [A5].Select

   ' Turn off the display of graphic objects and
   ' version borders in the template.
   [].ShowDrawLayer = False
   [].ShowVersionBorders = False

   ' Select the active area of the sheet.
   [].SelectAll

   ' Copy the selected range to the Clipboard
   ' as a bitmap image.
   .CopyToClipboard $BitMapFormat
   [A5].Select

   ' Log into the e-mail system.
   CurrentApplication.UserLogin "DKearns","newcastle"

   ' Send a mail message with the subject "Portfolio"
   ' to the mailing list MyTeam and attach the bitmap
   ' image of the selected range from the Clipboard.
   [].SendMail  "MyTeam",,"Portfolio",,,,$Clipboard

   ' Turn the display of graphic objects and version
   ' borders back on.
   [].ShowDrawLayer = True
   [].ShowVersionBorders = True
End Sub
```

Routing a range

TeamReview lets you use your e-mail system to send a range of workbook data to other 1-2-3 users. You can send a range to members of a team, collect their input, and have it returned to you automatically.

This example assumes that the user created a schedule template. Once each month, the user needs to get scheduling information from the manager of each department. This example routes the range that contains the schedule to a specified group of people. This example is written to be run from a button at the top of the sheet.

```
Sub RouteSchedule
   [A5].Select
   ' Turn off the display of graphic objects and
   ' version borders in the template.
   [].ShowDrawLayer = False
   [].ShowVersionBorders = False
   ' Select the active area of the sheet.
   [].SelectAll

   ' Log into the e-mail system.
   CurrentApplication.UserLogin "RSmith","taylor"

   ' Route the currently selected range to the department
   ' managers. Have changes mailed back as each person
   ' sends the range on to the next.
   [].Send "TeamLeaders",,"Schedule"

   ' Turn the display of graphic objects and
   ' version borders back on.
   [].ShowDrawLayer = True
   [].ShowVersionBorders = True
End Sub
```

Top tasks

This section describes some 1-2-3 tasks that you might want to automate and illustrates LotusScript solutions for them. This section describes the following tasks:

- Creating a custom @function
- Creating a custom menu
- Saving and restoring a view
- Changing labels to values
- Converting column and sheet letters to numbers
- Creating a cross-tabulation report
- Automatically saving all open workbooks
- Making global changes to a range

The code for each example is stored in a .123 file in the sample files directory. Each file contains sample data, as well as instructions for running the scripts it contains. The file name for each code example is given before each example.

Creating a custom @function

In the past, only add-in developers who used the 1-2-3 Add-In Toolkit could create their own @functions. Now, anyone can use LotusScript to write custom @functions for 1-2-3.

A custom @function is simply a LotusScript function—a procedure in a script with a name assigned to it that returns a value. Users use a custom

@function in the same way they use the @functions that are built into 1-2-3. They simply type the @function and its arguments into a cell.

To create a custom @function:

1. Choose Create - @Function.
2. Enter a name for the function in the “Name” box. Do not include the at sign (@) in the name.

   Follow the LotusScript function naming conventions when naming custom @functions. Do not use the name of an existing @function, macro keyword, or LotusScript keyword.
3. Click OK.

   1-2-3 displays the empty function in the Script Editor.
4. Enter statements that you want to execute when 1-2-3 invokes the function.
5. Close the Script Editor when you finish writing the function statements.

The following example creates an @function named @ELAPSED that calculates the number of hours, minutes, and seconds that have elapsed between two times, and displays the information as a label. @ELAPSED takes two arguments, a start time and an end time.

Note The text of this script is stored in DW07_S1.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

```
Function Elapsed (Start As Double, End As Double) As String
   Dim w As Double
   Dim x As Double
   Dim y As Double
   Dim z As Double

   ' Subtract start time from end time and extract
   ' the hours, minutes, and seconds from the
   ' resulting time number.
   w = End - Start
   x = Hour(w)
   y = Minute(w)
   z = Second(w)

   ' Display a label that contains the elapsed hours,
   ' minutes, and seconds.
   Elapsed = Str(x)&"Hr "&Str(y)&"Mn "&Str(z)&"Sc"
End Function
```

Creating a custom menu

One of the easiest ways to get users to run your scripts is to place them on a menu that you insert in the 1-2-3 main menu. The following example creates a menu that displays a list of custom @functions. When the user picks an @function from the menu, 1-2-3 displays it in a cell, with placeholders for the arguments. The code for the two custom @functions, @ISEVEN and @ISODD, is also included in this example.

If you want a menu to be available when the workbook that contains it is active, attach the menu script to the document Opened event:

1. Choose Edit - Scripts & Macros - Show Script Editor.

   The Script Editor appears.
2. In the Object drop-down box, select the name of the current file.
3. In the Script drop-down box, select Opened.

   The empty Opened sub appears in the Script Editor.
4. Enter statements in the sub that you want 1-2-3 to execute when the sub runs.
5. Save the script by saving the .123 file.

When you open the file, 1-2-3 automatically runs the script.

If you want a menu to be available at the start of each 1-2-3 session, attach the menu script to the document Opened event, then store the .123 file that contains the script in the “Automatically opened files” directory. At the start of each session, 1-2-3 opens all the files in the “Automatically opened files” directory in alphabetical order, words before numbers. Any scripts attached to the Opened event of a file will run when 1-2-3 opens the file. For more information about automatically opening files in 1-2-3, search on “Opening, files automatically” in the 1-2-3 Help Index.

Note The text of this script is stored in DW07_S2.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

```
'This definition appears in (Declarations):
Dim IsMenuSet as Integer

'Display the Functions menu when 1-2-3 opens this file.
Sub Opened(Source as Document)
Call SetMenu
   IsMenuSet = True
End Sub
```

```
Sub SetMenu
   'Set constants for menu and menu items.
   Const M_Functions ="F&unctions"
   Const M_IsOdd ="Is&Odd"
   Const M_IsEven ="Is&Even"

   ' Set constants for menu item long prompts.
   Const P_Functions ="Select a custom @function"
   Const P_IsOdd = "Returns True for an odd value"
   Const P_IsEven ="Returns True for an even value"

   ' Set constants for global function names.
   Const S_IsOdd ="PutIsOdd"
   Const S_IsEven ="PutIsEven"

   Dim FuncMenu As Menu
   Dim MainMenu As Menu

   SetMainMenu = CurrentApplication.MenuBar
   Set FuncMenu = CurrentApplication.NewMenu

   FuncMenu.MenuText = M_Functions
   FuncMenu.MenuPrompt = P_Functions

   ' This statement prevents duplicate menus if the
   ' Functions menu is already in the menu bar.
   MainMenu.DeleteItem M_Functions

   ' Add the Functions menu to the end of the
   ' 1-2-3 main menu.
   Call CurrentApplication.CurrentMenubar.AddMenu(-1, _
      FuncMenu)

   'Add the two menu items to FuncMenu.
   Call FuncMenu.AddItem(-1, M_IsOdd, P_IsOdd, _
      ThisDocument, S_IsOdd)
   Call FuncMenu.AddItem(-1, M_IsEven, P_IsEven, _
      AddItem(-1, ThisDocument, S_IsEven))
End Sub
```

Use the Preclose event to determine if the Functions menu is still displayed. If it is, use the Postclose event to remove the Functions menu.

```
Function Preclose(Source As Document, _
   P1 As Variant) As Variant
   ' If IsMenuSet = True, then the Functions menu is still
   ' displayed. Block the close.
   If IsMenuSet Then
      Preclose = $Block
   ' If IsMenuSet = False, then the Functions menu is not
   ' displayed. Close the file.
   Else
      Preclose = $Continue
   End If
End Function

Sub Postclose(Source As Document, P1 As Variant)
   Dim MainMenu As MenuBar
   Set MainMenu = CurrentApplication.CurrentMenuBar
   IsMenuSet = False
   ' If the Functions menu appears in the 1-2-3
   ' main menu, remove it.
   If MainMenu.GetItemText(-1) = "F&unctions" Then
      MainMenu.RemoveItem(-1)
   End If
   ' Close the file.
   Source.Close
End Sub

Sub PutIsOdd
   ' Enter the @ISODD @function in the current cell
   ' and leave 1-2-3 in Edit mode.
   .UpdateSheetDisplay = False
   Selection.Contents = "@ISODD(x)"
   SendKeys "{F2}"
   .UpdateSheetDisplay = True
End Sub

Sub PutIsEven
   ' Enter the @ISEVEN @function in the current cell
   ' and leave 1-2-3 in Edit mode.
   .UpdateSheetDisplay = False
   Selection.Contents = "@ISEVEN(x)"
   SendKeys "{F2}"
   .UpdateSheetDisplay = True
End Sub

' Returns 1 for an odd number; 0 for all other entries.
Function IsOdd (x As Integer) As Integer
   If x Mod 2 <> 0 Then
      IsOdd = 1
   Else
      IsOdd = 0
   End If
End Function

' Returns 1 for an even number; 0 for all other entries.
Function IsEven (x As Integer) As Integer
If x Mod 2 <> 0 Then
      IsEven = 0
   Else
      IsEven = 1
   End If
End Function
```

Saving and restoring a view

Often users work on the same 1-2-3 files, day after day. They must open those files at the start of every 1-2-3 session and then adjust the workbook window sizes and positions.

The scripts in this example perform the following tasks:

- The sub SaveDocGroup saves the names of all open workbooks, along with the size and position of their windows, and stores this information in a text file named WINLIST.LST in the default workbook directory.
- The sub OpenDocGroup opens all the files listed in the text file WINLIST.LST and restores their windows to their previous sizes and positions.

The user runs SaveDocGroup to save the current view, then runs OpenDocGroup to restore that view at the start of a 1-2-3 session.

Note The text of this script is stored in DW07_S3.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

```
' Opens the text file WINLIST.LST and enters the name
' of each active workbook and the size and position of
' its window on a separate line.
Sub SaveDocGroup
   Dim GroupFile as String
   Dim App As Application
   Dim WinLst As DocWindows
   Dim Win As DocWindow
   Dim l As Integer
   Dim File As Integer
   Dim FileName As String
   Dim WbNameL as String
   Dim WbNameS as String

   ' Create a collection of all open document windows.
   Set App = CurrentApplication
   Set WinLst = App.Windows
   MessageBox Str$(WinLst.Count) + " Open document windows"
   Set Win = WinLst.Open

   ' Open file WINLIST.LST for writing.
   On Error Goto OpenError   'Set error handler.
   File = FreeFile()
   FileName = GetWinListName()

   ' Remove the << >> that 1-2-3 automatically surrounds
   ' the workbook name with when it writes the name to the
   ' text file.
   WbNameL = Win.Document.Name
   WbLen = (Len(WbNameL)-4)
   WbNameS = Mid$(WbLenL,3,WbLen)
   Open FileName For Output As #File
   On Error GoTo 0           'Disable error handler.

   ' Write the names and window positions of all documents
   ' in the collection WinLst.
   For l = 0 To WinLst.Count-1
      Set Win = WinLst.Item(l)
      ' Enter the name of the document.
      Print #File, WbNameS; ",";
      ' Enter window position.
      Print #File, Win.Left; ","; Win.Top; ",";
      Print #File, Win.Height; ","; Win.Width;
      Print #File, 'Line feed

   Next
   Close #File

   ' Ends sub SaveDocGroup.
   ExitSave:
      Exit Sub

   ' Error routine for errors opening text file.
   OpenError:

      MessageBox "Open Error: "+Error$(Err)+" on "+GroupFile
      Close #File
      Resume ExitSave
End Sub
```

The user runs OpenDocGroup at the start of a 1-2-3 session to restore the view saved with SaveDocGroup.

```
' Opens the text file WINLIST.LST. Reads the name and
' window size and position of each workbook listed there
' and then opens the workbooks.
Sub OpenDocGroup
   Dim App As Application
   Dim File As Integer
   Dim Top As Integer
   Dim Lft As Integer
   Dim Wdth As Integer
   Dim Hght As Integer
   Dim RetVal As Integer
   Dim Fname As String
   Dim DocLst List As Document
   Dim FileName As String

   Set App = CurrentApplication
   ' Open file WINLIST.LST.
   ' Get next free file number.
   File = FreeFile()
   FileName = GetWinListName()
   On Error GoTo OpenError     ' Set error handler.
Open FileName For Input As #file
On Error Goto 0   ' Disable open error handler.

' Read each line of WINLIST.LST and open the
' workbooks listed.
Do While Not EOF(File)
   Input #file, Fname, Lft, Top, Hght, Wdth
   MessageBox Fname+","+Str$(Lft)+Str$(Top) _
      +Str$(Hght)+Str$(Wdth)

   If IsElement(DocLst(Fname)) Then
      ' If the workbook is already open, then
      ' open a new window for the same workbook.
      DocLst(Fname).NewDocWindow
      ' Reset the current window to its original size.
      CurrentWindow.Top = Top
      CurrentWindow.Left = Lft
      CurrentWindow.Height = Hght
      CurrentWindow.Width = Wdth
   Else
      ' The workbook is not open yet.
      ' Set error handler.
      On Error GoTo OpenDocumentError
      ' Open the workbook.
      App.OpenDocument Fname
      ' Disable error handler.
      On Error GoTo 0
      ' Add workbook name to list of opened documents.
      Set DocLst(Fname) = CurrentDocument
   End If

Loop
Close #file
.ActiveDocWindow.Restore
Exit Sub

' Error routine for opening text file
OpenError:

   MessageBox "Open error: "+Error$(Err) _
   +" on "+GroupFile+" (A group file may not have" _
      +" been saved yet.)",0,"Open Doc.Group"
   Exit Sub
   Resume 0
```

```
' Error routine for opening workbooks
OpenDocumentError:

   RetVal = MessageBox("Can't open Workbook" _
      +Fname, MB_OKCANCEL + MB_ICONINFORMATION, _
      "Open Doc.Group")
   If RetVal = IDOK Then
      'If user clicks OK, don't try to open workbook.

      Resume Next
      Else
         ' If user clicks Cancel, end sub.
         Close #File
         Exit Sub
      End If
End Sub
```

The function GetWinListName, which is used in the subs SaveDocGroup and OpenDocGroup, finds the text file WINLIST.LST.

```
Function GetWinListName() As String
   ' The name of the text file is WINLIST.LST.
   Const WinListName = "winlist.lst"
   ' WINLIST.LST should be in the 1-2-3 default directory.
   GetWinListName = CurrentApplication.DefaultPath + _
      WinListName
End Function
```

Changing labels to values

Data imported into 1-2-3 from text files or word-processing programs is often imported into 1-2-3 as labels (strings). In order to perform calculations on the data, you must first convert it to (numeric) values.

The following script converts the labels in a selected range to values. The script only converts labels that look like numbers; any other labels in the range remain unchanged. For example, the script converts the label '55 to the value 55, but does not change the label 'Budget. Similarly, the script converts a label that looks like a date number, such as '35704, but does not change a label that looks like a date format, such as 'Oct-97.

Note The text of this script is stored in DW07_S4.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

```
Sub LabelToValue
   Dim rs As RangeSelector
   Set rs = CurrentApplication.RangeSelector
   Dim r As Range
   ' Select a range to modify.
   Set r = rs.GetRange
   ForAll Cell In r.Cells
      If Cell.Contents <> " " Then
         Cell.Contents = Cell.CellValue
      End If
   End Forall
End Sub
```

Converting column and sheet letters to numbers

In 1-2-3, you will likely want to automate tasks that involve database tables or lookup tables. When you write scripts that work with tables, it is useful to be able to convert sheet and column letters to numbers and to convert the zero-based offset numbers returned by some LotusScript properties to their equivalent column and sheet letters.

This example converts column and sheet letters to numbers and offset numbers to column and sheet letters.

Note The text of these functions is stored in DW07_S5.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

The function NumToLetters converts a number from 0 through 255 to a column or sheet letter from A through IV.

```
Function NumToLetters(NumVal As Integer) As String
   If NumVal < 26 Then
      NumToLetters = Chr(NumVal + 65)
   Else
      NumToLetters = Chr((NumVal \ 26) + 64) & _
         Chr((NumVal Mod 26) + 65)
   End If
End Function
```

The function LettersToNum converts a letter from A through IV to a number from 0 through 255.

```
Function LettersToNum (Letters As String) As Integer
   If Len(Letters) > 1 Then
   LettersToNum = ((Asc(Ucase$(Letters)) - 64) _
      * 26) + (Asc(Ucase$(Right$(Letters,1))) - 64)
      Else
         LettersToNum = Asc(Ucase$(Letters)) - 64
   End If
End Function
```

Creating a cross-tabulation report

A cross-tabulation, or crosstab, table categorizes and summarizes database records. Where a database table has rows containing individual records, a crosstab table shows cells that summarize underlying records grouped by the fields you specify.

A crosstab table is a good tool for analyzing data with three or more variables. For example, use a crosstab table to present products by type, by quantity sold, and by sales representative.

The following example creates a crosstab table in a new sheet. Note that this example incorporates the NumToLetters and LettersToNum functions from the immediately preceding example in this section.

Note The text of this script is stored in DW07_S6.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

```
Sub CrossTab
   Dim SelectRange As String
   Dim Range1 As Range
   Dim Col1 As String
   Dim Col2 As String
   Dim ColNum1 As Integer
   Dim ColNum2 As Integer
   Dim Row1 As Integer
   Dim Row2 As Integer
   Dim Sheet1 As String
   Dim Sheet2 As String
   Dim Temp1 As Integer
   Dim FieldList List As String
   Dim ListString As String
   Dim ColHeading As String
   Dim RowHeading As String
   Dim ActionField As String
   Dim CTAction As String
   Dim FunctionName As String
   Dim FunctionDesc As String
   Dim DefField As String

   ' Constants that determine the placement of input boxes
   Const XPos = 3000
   Const YPos = 3000

   ' Error message constants. Text can be
   ' changed or translated.
   Const SmallRangeError = "Database range must contain" _
      +" at least two rows and three columns."
```

```
Const No3DError = "Database range cannot span" _
   +" worksheets."
Const BlankFieldError = "Column headings cannot" _
   +" be blank, numbers, or formulas."
Const InvalidField = "Invalid field name." _
   +" Choose a field name from the list."
Const InvalidOp = "You specified an invalid" _
   +" operation! Choose an operation from the list."

' Input box constants. Text can be changed or translated.
Const RowPrompt = "Choose a row heading field from" _
   +" the following fields: "
Const RowTitle = "Row Heading"
Const ColPrompt = "Choose a column heading field from" _
   +" the following fields: "
Const ColTitle = "Column Heading"
Const SumFieldPrompt = "Choose a summary field" _
   +" from the following fields: "
Const SumFieldTitle = "Summary Field"
Const SummaryPrompt = "Choose the summary operation: "
Const SummaryTitle = "Crosstab Data Options"

' Operator constants
Const SumName = "Sum"
Const SumFunc = "Dsum"
Const SumDesc = "Total"
Const AvgName = "Average"
Const AvgFunc = "Davg"
Const AvgDesc = "Average"
Const CntName = "Count"
Const CntFunc = "Dcount"
Const CntDesc = "A count of"
Const MaxName = "Maximum"
Const MaxFunc = "Dmax"
Const MaxDesc = "Maximum"
Const MinName = "Minimum"
Const MinFunc = "Dmin"
Const MinDesc = "Minimum"

' String constants. Text can be changed or translated.
Const CTDesc1 = "Crosstab table for "
Const CTDesc2 = " by "
Const CTDesc3 = " and "

SelectRange = .CoordinateString

' Check to see if selection is a single cell.
If InStr(1, SelectRange, "..") = 0  Then
```

```
      ' Display error for single-cell selection.
      MessageBox SmallRangeError
      Exit Sub
End If

' Extract sheet letters and convert them to numbers.
Sheet1 = Left$(SelectRange, InStr(1, SelectRange, _
   ":") - 1)
Sheet2 = Mid$(SelectRange, InStr(1, SelectRange, _
   "..") + 2, InStr(4, SelectRange, ":")  - _
   InStr(1, SelectRange, "..") -2)

' Check for multiple-sheet selection.
If LettersToNum(Sheet2) - LettersToNum(Sheet1) _
   <> 0  Then
   ' Display error for multiple-sheet selection.
   MessageBox No3DError
   Exit Sub
End If

' Extract column letters and convert to numbers.
Col1 = Mid$(SelectRange, InStr(1, SelectRange, _
   ":") + 1, 1)
If IsNumeric (Mid$(SelectRange, InStr(1, _
   SelectRange, ":") + 2, 1 )) = 0 Then
   Col1 = Col1 & Mid$(SelectRange, InStr(1, _
      SelectRange, ":") + 2, 1)
End If

Col2 = Mid$(SelectRange, InStr(4, SelectRange, _
   ":") + 1, 1)
If IsNumeric (Mid$(SelectRange, InStr(4, _
   SelectRange, ":") + 2, 1 )) = 0 Then
   Col2 = Col2 & Mid$(SelectRange, InStr(4, _
      SelectRange, ":") + 2, 1)
End If

' Sort columns.
If LettersToNum(Col2) > LettersToNum(Col1) Then
   ColNum1 = LettersToNum(Col1)
   ColNum2 = LettersToNum(Col2)
Else
   ColNum1 = LettersToNum(Col2)
   ColNum2 = LettersToNum(Col1)
End If
```

```
' Extract row numbers.
Row1 = CInt(Mid$(SelectRange, InStr(1, SelectRange, _
   ":") + 1 + Len(Col1),  InStr(1, SelectRange, _
   "..") - InStr(1, SelectRange, ":") - 1 - Len(Col1)))
Row2 = CInt(Mid$(SelectRange, InStr(4, SelectRange, _
   ":") + 1 + Len(Col2),  Len(SelectRange) _
  - InStr(4, SelectRange, ":")  - Len(Col2)))

If Row2 < Row1 Then
   Temp1 = Row1
   Row1 = Row2
   Row2 = Temp1
End If

' Check that the range contains at least two
' rows and three columns.
If (Row2 - Row1) < 1 Or (ColNum2 - ColNum1) < 2 Then
   ' If the range is smaller than two rows by
   ' three columns, display error message.
   MessageBox SmallRangeError
   Exit Sub
End If

' Get field names and check them for spaces or formulas.
Dim y As Integer
For x = ColNum1 To ColNum2
   y = x
   Set Range1 = Bind(Sheet1 & ":" & NumToLetters(y) _
      & Cstr(Row1))
   If (Range1.CellValue <> Mid$(Range1.Contents,2) _
      Or Range1.CellValue = "") Then
      ' Display error for field names that contain
      ' spaces or formulas.
      MessageBox BlankFieldError
      Exit Sub
   End If
   FieldList(y) = Range1.CellValue
Next

' Choose row headings.
ForAll z In FieldList
   ListString = ListString & ", " & z
End ForAll
ListString = Mid$ (ListString, 3)

RowHeading = "%^#"
DefField = Left$(ListString, _
   InStr(1, ListString, ",") - 1)
Do While InStr(1, ListString, RowHeading, 1) = 0
   RowHeading = InputBox$(RowPrompt & ListString, _
      RowTitle, DefField , XPos, YPos)
      ' End script execution if the user clicks Cancel.
   If RowHeading = "" Then
      Exit Sub
   End If
   If InStr(1, ListString, RowHeading, 1) = 0 Then
      ' Display error for wrong field name.
      MessageBox InvalidField
   End If
Loop

' Choose column headings.
ListString = ""
ForAll w In FieldList
   If w <> RowHeading Then
      ListString = ListString & ", " & w
   End If
End ForAll
ListString = Mid$(ListString, 3)

ColHeading = "%^#"
DefField = Left$(ListString, InStr(1, ListString, _
   ",") - 1)
Do While InStr(1, ListString, ColHeading, 1) = 0
   ColHeading = InputBox$(ColPrompt & ListString, _
      ColTitle, DefField, XPos, YPos)
   ' End script execution if the user clicks Cancel.
   If ColHeading = "" Then
      Exit Sub
   End If
   'Display error for wrong field name.
   If InStr(1, ListString, ColHeading, 1) = 0 Then
      MessageBox InvalidField
   End If
Loop

' Choose action field.
ListString = ""
ForAll q In FieldList
   If (q <> RowHeading And q <> ColHeading) Then
      ListString = ListString & ", " & q
   End If
End ForAll
ListString = Mid$(ListString, 3)
ActionField = "%^#"

If InStr(1, ListString, ",") = 0 Then
   DefField = ListString
```

```
Else
   DefField = Left$(ListString, InStr(1, _
      ListString, ",") - 1)
End If

Do While InStr(1, ListString, ActionField, 1) = 0
   ' Display Summary Operation input box.
   ActionField = InputBox$(SumFieldPrompt & _
      ListString, SumFieldTitle, DefField, XPos, YPos)
   'End script execution if the user clicks Cancel.
   If ActionField = "" Then
      Exit Sub
   End If
   ' Display error for wrong field name.
   If InStr(1, ListString, ActionField, 1) = 0 Then
      MessageBox InvalidField
   End If
Loop

ListString = "Sum, Average, Count, Min, Max"
CTAction = "&*%$"
Do While InStr(1, ListString, CTAction, 1) = 0
   CTAction = InputBox$(SummaryPrompt & ListString, _
      SummaryTitle, "Sum", XPos, YPos)
   ' End script execution if the user clicks Cancel.
   If CTAction = "" Then
      Exit Sub
   End If
   If InStr(1, ListString, CTAction, 1) = 0 Then
      ' Display error for wrong operation.
      MessageBox InvalidOp
   End If
Loop

' Set operation name and string name
' for title, based on CTAction.
Select Case Ucase$ (CTAction)
Case SumName :FunctionName = SumFunc
   FunctionDesc = SumDesc
Case AvgName :FunctionName = AvgFunc
   FunctionDesc = AvgDesc
Case CountName :FunctionName = CountFunc
   FunctionDesc = CountDesc
Case MaxName :FunctionName = MaxFunc
   FunctionDesc = MaxDesc
Case MinName :FunctionName = MinFunc
   FunctionDesc = MinDesc
End Select
```

```
' Insert a new sheet after the current sheet.
' Crosstab table appears on new sheet.
.NewSheet  $After, 1, True

' Freeze screen to prevent flashing while
' 1-2-3 builds the crosstab table.
CurrentApplication.UpdateSheetDisplay = False

' Get unique list of row heading entries with
' 1-2-3 Classic command /Data Query Unique.
[A5].Contents = RowHeading
[A1].Contents = "/dqri" & SelectRange & "~oa5~uq"
[A1].MacroRun
[A5].Cut

' Get unique list of column heading entries with
' 1-2-3 Classic command /Data Query Unique and
' then transpose the data to display across the rows.
[B5].Contents = ColHeading
[A1].Contents = "/dqri" & SelectRange & _
   "~ob5..b" & CStr(Row2 - Row1 +4) & "~uq"
[A1].MacroRun

Set Range1 = Bind("b6..b" & CStr(Row2 - Row1 +4))
Range1.Transpose[B5]
Range1.Cut
[A1].Cut

' Enter the title for the table in cell A4.
[A4].Contents = CTDesc1 & FunctionDesc & " " & _
   ActionField & CTDesc2 & RowHeading & CTDesc3 & _
   ColHeading
[A4].Font.Bold = 1

' Set up the what-if table with two variables.
[A1].Contents = RowHeading
[B1].Contents = ColHeading
[A5].Contents = "@" & FunctionName & _
"(" & SelectRange & ",""" & ActionField & """,a1..b2)"

' Create the crosstab table.
[C1].Contents = _
   {GoTo}a5~/dtr/dt2.{End}{d}...{End}{r}~a2~b2~"
[C1].MacroRun
```

```
   ' Cleanup work: Delete macros from sheet and
   ' turn sheet display back on.
   [A1..A3].DeleteRows $Full
   [A2].Cut
   [A1].Select
   CurrentApplication.UpdateSheetDisplay = True
Exit Sub

'Error routine for any other error not previously
'specified in the CrossTab sub.
GenError:
   Messagebox Error$()
   .DeleteSheet
End Sub
```

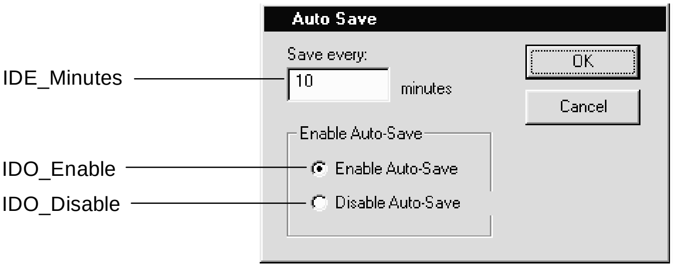
Automatically saving all open workbooks

The following example automatically saves all open workbooks when a specified number of minutes elapses. This example displays a dialog box that lets users specify the following:

- A number of minutes, from 1 through 1440 (24 hours)
- Whether to turn automatic file saving on or off

Note The text of this script is stored in DW07_S7.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

The sub AutoSave, in (Globals), displays the AutoSave dialog box. The names of the dialog box controls, which are used throughout the script, are shown in the following illustration:



Save every: 10 minutes

Enable Auto-Save

● Enable Auto-Save

○ Disable Auto-Save

```
Sub AutoSave
   AutoSaveDlg.Show
End Sub

' Close the dialog box if the user clicks Cancel.
Sub Click(Source As LotusCommandButton)
```

```
   ' Do nothing; close the dialog box.
   Call Source.Parent.Close()
End Sub
```

If the user clicks the OK button, 1-2-3 runs the InitAutoSave sub and then closes the dialog box:

```
Sub Click(Source As LotusCommandButton)
   ' Start the timer and close the dialog box.
   Call InitAutoSave()
   Call Source.Parent.Close()
End Sub
```

The following scripts are in the (AutoSaveDlg Globals) object, which contains global scripts for the dialog box object AutoSaveDlg. To see these scripts in the IDE, click the arrow to the left of the AutoSaveDlg object in the Object drop-down box and then select the (AutoSaveDlg Globals) object. The scripts for the object are listed in the Script drop-down box.

```
' The constant definitions appear in (Declarations):
' Default elapsed time is 10 minutes.
Const SaveTimer = 10
' Minimum elapsed time is 1 minute.
Const MinTimer = 1
' Maximum elapsed time is 1440 minutes (24 hours).
Const MaxTimer = 1440   ' Maximum elapsed time = 1440
                        ' minutes (24 hours)
Const AutoSavePoll = 1

Sub InitAutoSave
   Dim PollTimer As Double

   ' Enable the Poll event if the user selected
   ' "Enable Auto-Save," or disable the Poll event if
   ' the user selected "Disable Auto-Save."
   If AutoSaveDlg.IDO_Enable.Value Then
      'Get timer value.
      PollTimer = Val(AutoSaveDlg.IDE_Minutes.Caption)

      'Check for a valid value for the timer.
      If PollTimer < MinTimer Or PollTimer > _
         MaxTimer Then
         MessageBox "The Auto-Save timer is invalid", _
            MB_OK, "Invalid Timer"
         Exit Sub
      End If

      ' Multiply the value the user entered in
      ' the dialog box by 60 * 1000 to convert it
      ' into milliseconds.
      PollTimer = PollTimer * 60 *1000
      ' Start the timer.
      Call ThisDocument.StartPoll(AutoSavePoll, _
         PollTimer, 0)
   Else
      Call ThisDocument.EndPoll(AutoSavePoll)
   End If
End Sub
' This Poll1 event script for the document object saves
' the open files.
Sub Poll1(Source As Document)
   ' Save any open workbooks that have been modified
   ' since the last save.
   ForAll Doc In CurrentApplication.Documents
      If Doc.Changed Then
         Call Doc.Save()
      End If
   End ForAll
End Sub
```

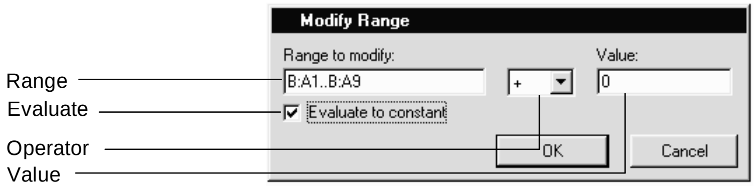
Making global changes to a range

You can write a script that makes it easy for the user to make global changes to an entire range. For example, suppose the user frequently imports numeric data and always makes the same mathematical changes to it. You can write a script that automates those changes for a specified range.

The following example displays a dialog box that lets the user select a range and then choose from a number of changes to apply to the entire range. The user can also decide if the changes result in formulas or values. For example, suppose the user chooses to multiply each value in the range A1..A10 by 5. If cell A3 contains the value 100, the script can store the result either as the formula 100*5 or as the value 500.

Note The text of this script is stored in DW07_S8.123 in the sample files directory. To view the script, open the file by choosing File - Open from the 1-2-3 main menu and then choose Edit - Scripts & Macros - Show Script Editor to display the IDE.

The sub ModifyRange, in (Globals), displays the Modify Range dialog box. The names of the dialog box controls, which are used throughout the script, are shown in the following illustration:



Range to modify: B:A1..B:A9

Value: 0

☑ Evaluate to constant

```
Sub ModifyRange
   ModifyRangeDlg.Show
End Sub

' The following script is executed when 1-2-3
' loads the ModifyRangeDlg dialog box.
Sub Load(Source As Lotusdialog)
   Dim RgText As LotusTextBox, ValText As LotusTextBox
   Dim Cbox As LotusComboBox

   ' Get the selected range.
   Set RgText = Source.Range

   ' Display the address of the currently selected
   ' range in the "Range to modify" box.
   RgText.Text = CurrentDocument.Selection.Allnames(0)

   ' Get the the Operation control.
   Set Cbox = Source.Operation

   ' Add mathematical operators to drop-down box.
   Cbox.AddItem("+")
   Cbox.AddItem("-")
   Cbox.AddItem("*")
   Cbox.AddItem("/")
   Cbox.SelectItem(0)
   Cbox.Refresh

   ' Get the Value control.
   Set ValText = Source.Value

   ' Display an initial value in the ValText text box.
   ValText.Text = "0"
End Sub

' Close the dialog box if the user clicks Cancel.
Sub Click(Source As LotusCommandButton)
   ' Do nothing; close the dialog box.
   Source.Parent.Close
End Sub
```

```
' Perform the specified calculation on the selected
' range if the user clicks OK.
Sub Click(Source As LotusCommandButton)
   Dim Sel As Variant
   Dim Oper As LotusComboBox
   Dim Value As LotusTextBox
   Dim EvalCheck As LotusCheckBox

   ' Get the value of the "Range to modify" text box.
   Set Sel = Selection

   ' Get the value of the Operator drop-down box.
   Set Oper = Source.Parent.Operation

   ' Get the specified value from the Value text box.
   ' There is no check for a valid numeric value.
   Set Value = Source.Parent.Value

   ' Get the value of the "Evaluate to constant" check box.
   Set EvalCheck = Source.Parent.Evaluate

   Call DoModifyRange(Sel, Oper.Caption, _
      Value.Text, EvalCheck.Value)

   'Close the dialog box.
   Source.Parent.Close
End Sub
```

The following script is in the (ModifyRangeDlg Globals) object, which contains global scripts for the dialog box object ModifyRangeDlg. To see these scripts in the IDE, click the arrow to the left of the ModifyRangeDlg object in the Object drop-down box and then select the (ModifyRange Globals) object. The scripts for the object are listed in the Script drop-down box.

```
Sub DoModifyRange(Sel As Variant, Oper As String, _
   Value As String, Eval As Variant)

   Dim RetVal As Integer
   Dim Dtype As Integer

   ' Loop through all rows and columns of selected range
   ' and modify the contents of the cells.
      ForAll X In Sel.Cells
         Dtype = DataType(X.CellValue)
         ' Make sure the current cell contains a value.
         If X.Contents <> "" And Ctype >= _
           V_INTEGER And Dtype < V_STRING Then
             X.Contents = "(" + X.Contents + ")"  _
               + oper + " " + Value
```

```
            ' If user checked "Evaluate to a constant,"
            ' then replace the formula with its value.
            If Eval Then
               X.Contents = X.CellValue
            End If
         End If
      End ForAll
   End ForAll

End Sub
```

Chapter 8
Using LotusScript in Approach

Writing scripts in Approach

LotusScript is an object-oriented programming language for automating tasks in applications you develop in Approach. LotusScript is more powerful and flexible than the Approach macro language, but it also requires that you have some basic programming skills.

Some of the tasks that you can accomplish with LotusScript in Approach include the following:

- Triggering the execution of scripts in response to user actions (clicking or double-clicking the mouse, pressing a key)
- Changing the attributes (color, size, position, visibility) of a text block or other display elements in views
- Displaying or manipulating Approach dialog boxes
- Quickly searching or making global changes to data in large databases
- Automating the display and modification of data in views
- Incorporating OCXs (OLE controls) into an application

Automating tasks without scripts

When you are planning to automate Approach tasks, keep in mind some of the other tools Approach provides for automating tasks. In some cases, there may be a better method than writing a script for accomplishing a particular task. For example, creating a named find lets you search for records according to the find conditions you define. Then, by saving the conditions, you can easily repeat the search anytime. Creating a named find in Approach is easier and more efficient than writing a script in LotusScript to do the same task.

Another example of a feature that automates tasks is Drill-down to Data, which allows you to select data in a crosstab or chart and view the details behind that data. When you display information in a crosstab or chart, you see grouped data and calculated summaries, totals, counts, averages, and so on. Drill-down reveals the record values that make up the groups or calculated values and displays the records in a worksheet.

These complex but common tasks can be accomplished with LotusScript, but why write a script if Approach has already automated the task for you?

Macros are another way of automating frequently performed tasks and are easily built using point-and-click functionality. Some of the tasks you can easily automate using macros include the following:

- Switching from one Approach view to another in an Approach document
- Changing data in a field for a small-to-medium set of records
- Importing or exporting records
- Switching between records
- Displaying a box containing a message and offering the user a choice of actions
- Finding records according to user-specified input

Macros have the advantage of being very easy to build. LotusScript has the advantages of speed and flexibility, especially when you want to process large amounts of data in a database.

Using scripts and macros together

Your application may use both scripts and macros for automation. When executing a script or macro, Approach does not give preference to one or the other. To avoid a situation that requires Approach to run scripts or macros in a particular order, follow these guidelines:

- Don’t trigger both a script and a macro from the same event.
- To control the order of operations, call a script from inside the body of a macro.

For example, if you want both a script and a macro to run when users close an application, define the macro to run when the application is closed and call the script from inside the macro.

Information for upgraders

This release of Approach introduces the ability to create scripts by recording user actions. Creating a transcript of an action is described later in this chapter in “Recording scripts in Approach.”

This release also contains new classes for creating finds and sorts. The Find class allows you to automate creating a found set. For more information, see “Find class” later in this chapter.

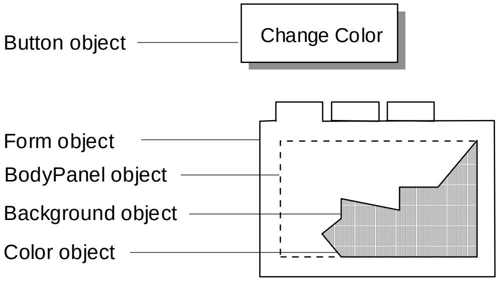
The Approach object model

You must understand the Approach object model before you can efficiently write scripts. In particular, understanding how Approach objects are related to one another in the containment hierarchy helps you use dot notation to identify objects.

Before continuing to read this section, you should be familiar with LotusScript classes and objects, containment and inheritance between LotusScript classes, and dot notation. For this information, see Chapter 2.

Like all LotusObjects, objects in Approach have properties, methods, and events. Properties are characteristics that describe an object's state. Methods are subs or functions that can be performed by the object. Events represent external actions performed on objects by users, the application, or the operating system.

For example, suppose you are writing a script to change the color of a form when the user clicks a button. You associate a script that changes the color of the form with the Click event for the button. As shown here, the form itself contains other objects that come into play as you write the script.

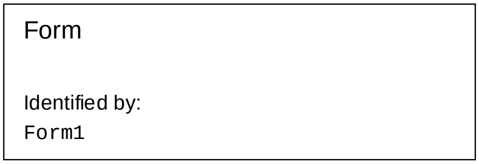


The following sequence describes the objects that are involved in changing the color of the form and the dot notation used to identify each object.

- The form is represented by an object.

  It is an instance of the Form class. The name of the Form object is Form1. Although the form probably wasn't created in a script, a Form object exists representing the actual form on the screen.

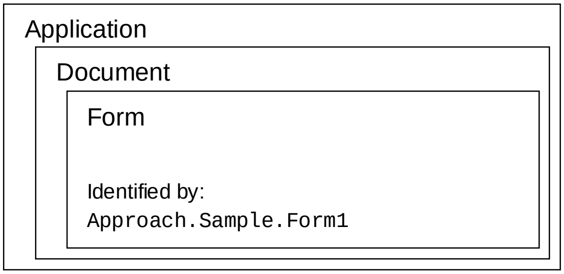
  You need to know about the form as an object so that you can work with the objects it contains and manipulate the characteristics of the form itself.



Specifying the Form object first ignores two other important objects:

- A Document object, in this case, SAMPLE.APR
- An Application object, or the session of Approach in which the .APR file is open

Considering these objects, the identification of the form changes as shown:

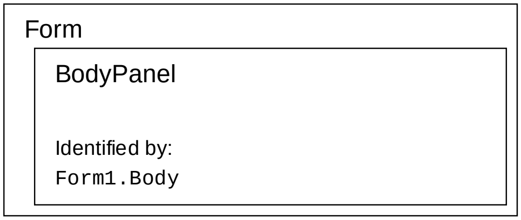


For the moment, however, consider the Form object as the top level of the hierarchy.

- The body of the form is represented by a separate object.

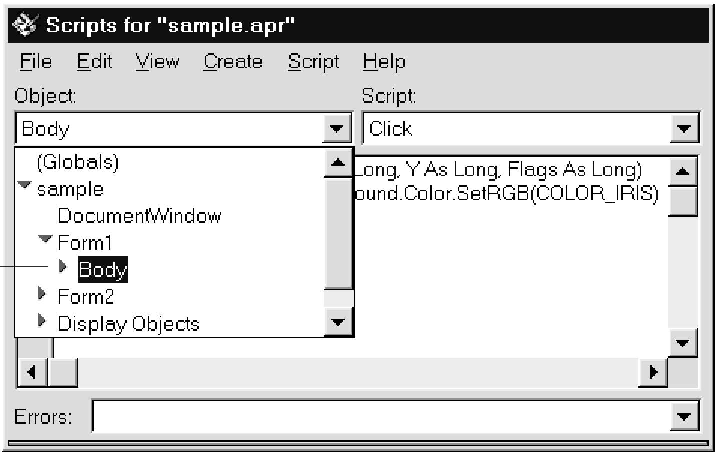
  It is an instance of the BodyPanel class, which is contained by the Form class, through the expanded property Body. Approach automatically creates a BodyPanel object when it creates the form.

  You need to know about the BodyPanel object because you control characteristics of the form, such as its color, through the BodyPanel object.



Expanded properties are properties of an object that are created automatically when a related object is created. In this example, when you created Form1, an expanded property identifying the BodyPanel object of the form is also created as part of Form1. The section "Approach containment hierarchy" later in this chapter describes expanded properties in more detail.

When you create scripts in Approach, the Integrated Development Environment (IDE) shows you where the BodyPanel object (identified by Body) appears in the containment hierarchy.

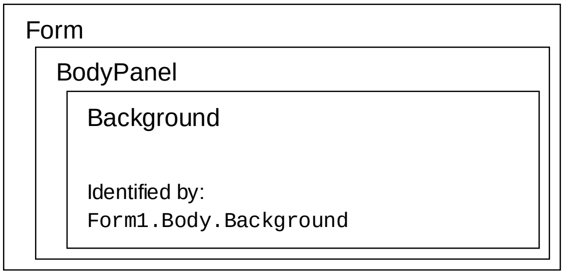


```
ong, Y As Long, Flags As Long)
ound.Color.SetRGB(COLOR_IRIS)
```

The Object listing shows each object in the containment hierarchy.

- The background of the form is represented by a separate object.

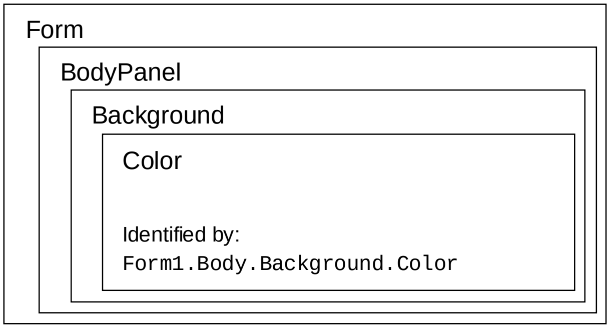
  It is an instance of the Background class, which is contained by the BodyPanel class, through the expanded property Background. Approach automatically creates a Background object named Background when it creates an object with a background.



- The color of the background is represented by a separate object.

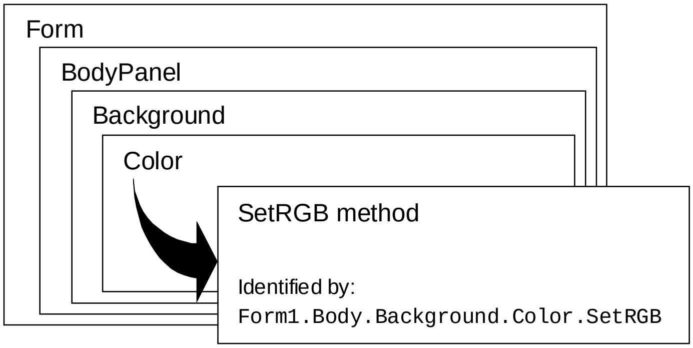
  It is an instance of the Color class, which is contained by the Background class, through the property Color. This object represents the characteristics of the color of the background of the body of the form.

  The Color class has many characteristics, including descriptions of the red, blue, and green components that make up the final color of an object. Approach automatically creates a Color object when it creates an object that can have colors.



- You can change the color of the background of the body of the form using a method of the Color object.

  The Color class includes the SetRGB method, which changes the color of an object.



To change the color of the form, then, you call the SetRGB method as follows, with an Approach LotusScript constant describing the new color:

```
SetRGB(COLOR_IRIS)
```

For information about choosing and setting colors using the Color class, search on “Classes (LotusScript)” in the Approach Help Index.

At this point, recall that the form is not at the top of the hierarchy. Approach has a predefined global product variable, CurrentDocument, that specifies the Approach executable (Application object) and .APR file (Document object) that are currently in use. Using this variable to indicate the part of the hierarchy above the form, the statement used to call the SetRGB method is as follows:

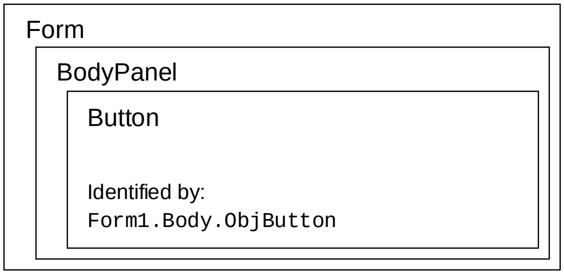
```
Call CurrentDocument.Form1.Body.Background.Color.SetRGB _
   (COLOR_IRIS)
```

Next, you must define the button the user clicks to change the color of the form and the event that triggers the change.

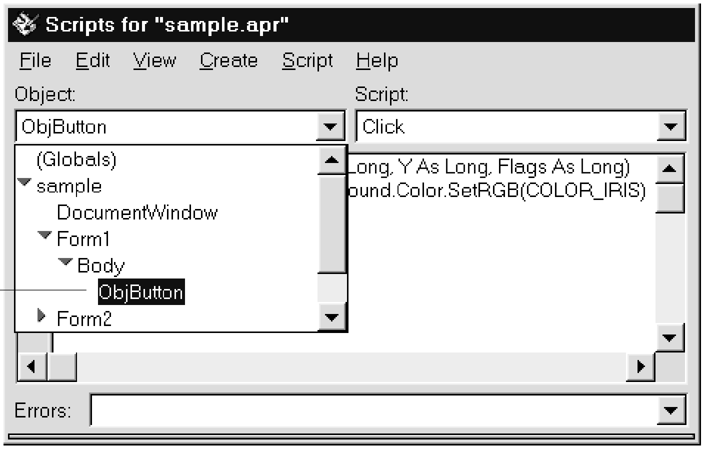
- The button that the user clicks is represented by an object.

  Like the background of the form, the button is identified by its location in the hierarchy.

  The button is an instance of the Button class. The name of the Button object is ObjButton, by default. The Button object is contained by the BodyPanel class, through an expanded property that corresponds to the name of the Button object.



In the IDE, you can see where the button appears in the containment hierarchy.

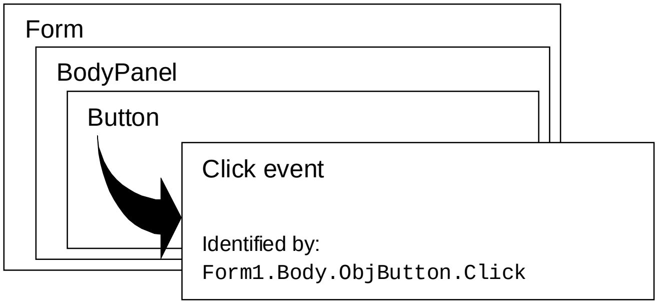


```
Long, Y As Long, Flags As Long)
ound.Color.SetRGB(COLOR_IRIS)
```

The Object listing shows each object in the containment hierarchy.

- The button has a Click event that represents the user's act of clicking the button.

  You associate the statement that calls the SetRGB method with this Click event. The statement runs when the user clicks the button.



`Form1.Body.ObjButton.Click`

Associate the statement with the Click event by entering the statements inside a sub for the Click event. To do this, choose ObjButton from the Object drop-down box in the IDE, make sure the Click event is showing in the Script drop-down box, and enter the statement after the Sub Click line, as follows:

```
Sub Click(Source As Button, X As Long, Y As Long, _
   Flags As Long)

Call CurrentDocument.Form1.Body.Background.Color.SetRGB _
   (COLOR_IRIS)

End Sub
```

The preceding example illustrates the way that Approach objects are related to one another in a hierarchy. It further illustrates the way that you identify objects and their properties, methods, and events by their location in the hierarchy.

The following sections go on to describe the hierarchy of Approach objects in more detail. One goal of this description is to make you familiar enough with these relationships that after some study and use, you'll be able to write scripts without having to look up the hierarchy of classes each time you need to refer to an object.

The sections describe Approach objects in the following order:

- Approach containment hierarchy
- Approach class inheritance
- Predefined global product variables that help you specify Approach objects
- The most important Approach classes and their members
- Examples of how you specify Approach objects in scripts

Approach containment hierarchy

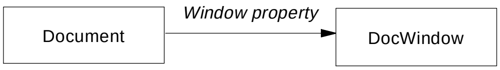
For general information about classes and containment, see "Containment" in Chapter 2.

The containment hierarchy for Approach begins at the top with the Application class and works its way through the familiar elements of the Approach interface: from .APR files to views to panels to display elements such as field boxes and text blocks. The relationship between container and contained object is always established through a property of the container. If you don't know the relationship between two objects, look for a property of the container in one of the following three configurations:

- The property is included in the definition of the container class.

  The most simple case allows you to know the containment relationship from the description of an object. The property name is part of the class definition. You can find the property listed in the Browser, and by selecting the property and pressing **F1**, you can see a description of the property in Help.

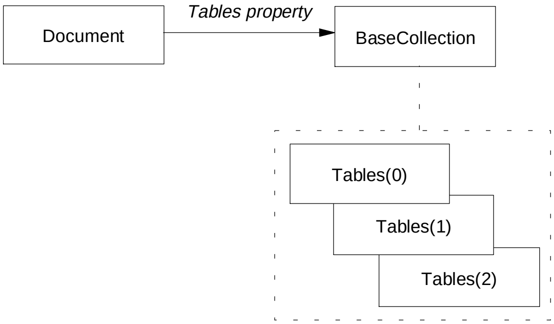
  For example, the DocWindow class is contained by the Document class through the property Window.



- The property identifies a collection (or array) of the contained objects, where the object you seek is one of the objects in the collection.

  This case also allows you to know the containment relationship by looking at the class description. You must also know the relationship between the object you seek and the collection. Identify the object by referring to an element of the collection.

  For example, all of the tables for an .APR file (Document object) are in a collection, or array, of Table objects. The collection is a BaseCollection object and is contained in the Tables property of the Document object. Each Table object is an element of the BaseCollection object. To identify a Table object, you specify which element that Table object is in the BaseCollection using an index.



You can manipulate the BaseCollection object as you would any other Approach object. For example, the following script creates an object x that represents the collection of tables in MyDocument:

```
Dim x As BaseCollection
Set x = MyDocument.Tables
```

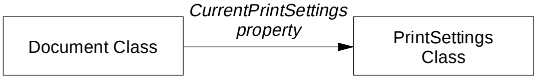
LotusScript allows you to access each element of the collection using notation built into the language. For example, the table OrderData is the second table in the collection (numbered from 0). Access OrderData directly as follows:

```
Dim y As Table
Set y = MyDocument.Tables(1)
```

The elements of the BaseCollection object are numbered from 0.

- The property doesn't appear as part of the container until the contained object is created.

  In this case, the property defining the containment relationship is an expanded property of the container. The property takes the name of the contained object. For example, a Form object named OrderEntryForm is contained by a Document object named Orders through the expanded property OrderEntryForm.

  When you cannot name the contained object because it has no Name property, the expanded property takes the class name. For example, a Background object is contained through the expanded property Background.

  The BodyPanel object is an exception to this rule; although the class is named BodyPanel, it is contained through the expanded property Body.

Some objects can be identified by more than one property. The expanded property is often more convenient in these cases. For example, you can identify a Document object through its location in a collection, as follows:

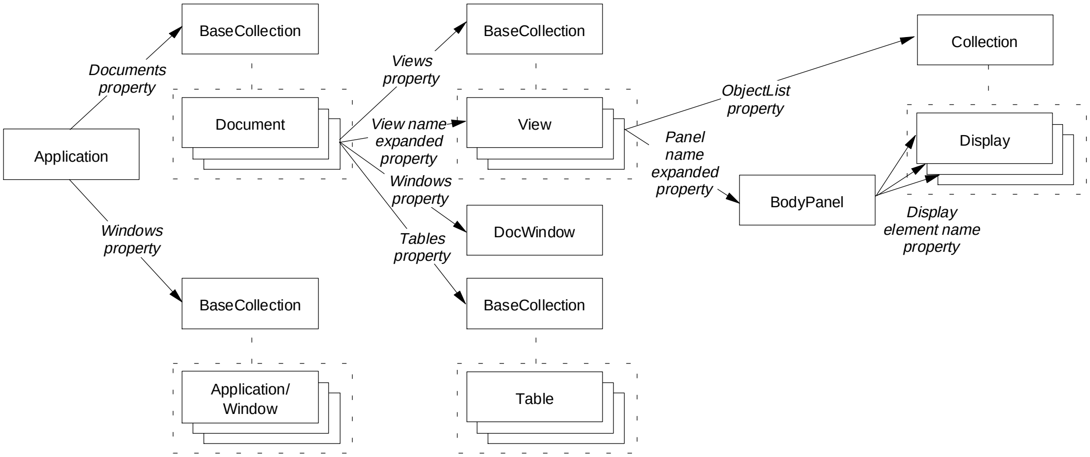
```
Dim x As Document
Set x = CurrentApplication.Documents(1)
```

You can more easily identify the Document, however, by using the name of the Document object, which is an expanded property of the Application object:

```
Dim x As Document
Set x = CurrentApplication.Orders
```

Containment hierarchy diagram

The following diagram shows the containment relationships for many Approach classes. The Application class is the top of the hierarchy, shown here at the far left. The classes that are not shown are not contained by other classes.

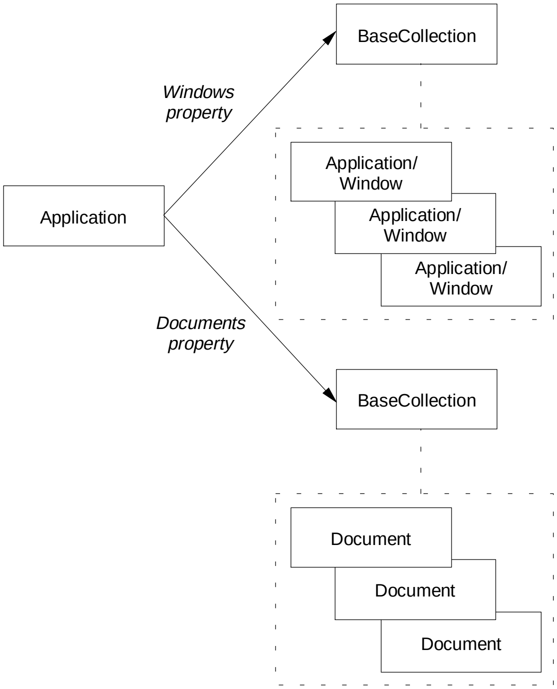


Application class as a container

The Application class in Approach represents the Approach session invoked from an executable.

Note In this chapter, the term “application” refers to a Document object (an .APR file) that you are using or developing. The term “Application object,” however, refers to Approach itself rather than your application. Expect the term “Approach” when the text refers to an Application object, and “application” or “.APR file” when the text refers to a Document object.

An Application object can contain two kinds of objects: ApplicationWindow objects and Document (.APR file) objects. In both cases, the contained objects are part of a BaseCollection object identified through a property of the Application class. The following illustration shows these containment relationships:



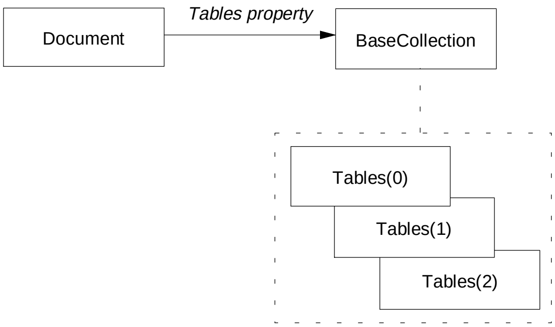
To identify a document relative to the current application, specify the index of the Document object in the BaseCollection object identified by the Documents property, as follows:

```
CurrentApplication.Documents(0)
```

Document class as a container

The Document class represents an application (an .APR file) running in Approach.

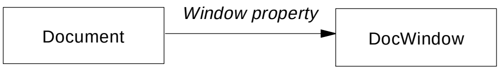
A Document object can contain three kinds of objects: Table objects, a DocWindow object, and objects representing Approach views. Table objects are related to a Document object through the Tables property, which identifies all of the tables associated with the .APR file in a collection (or array) of objects called a BaseCollection. Each Table object is an element of the collection.



To identify a Table object relative to the current document, specify the index of the Table object in the BaseCollection object identified by the Tables property, as follows:

```
CurrentDocument.Tables(0)
```

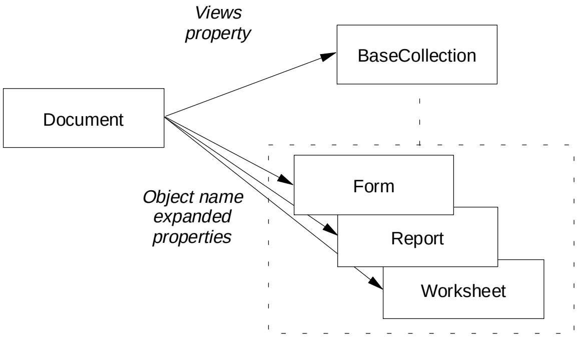
The DocWindow object is related to the Document object through a property. The DocWindow object represents the window inside the Approach window that contains the views for a specific Document object (.APR file). There is only one DocWindow object for each Document; the DocWindow object is identified through the Window property.



To identify a document window relative to the Document object, you specify the DocWindow object through the Window property:

```
CurrentDocument.Window
```

Objects representing Approach views are related to Document objects both through a collection class and through expanded properties, as shown:



Using these containment relationships, you can identify a view in either of the following ways:

- Using the view name as an expanded property

  `CurrentDocument.DataEntryForm`

- Using the collection through the Views property

  `CurrentDocument.Views(0)`

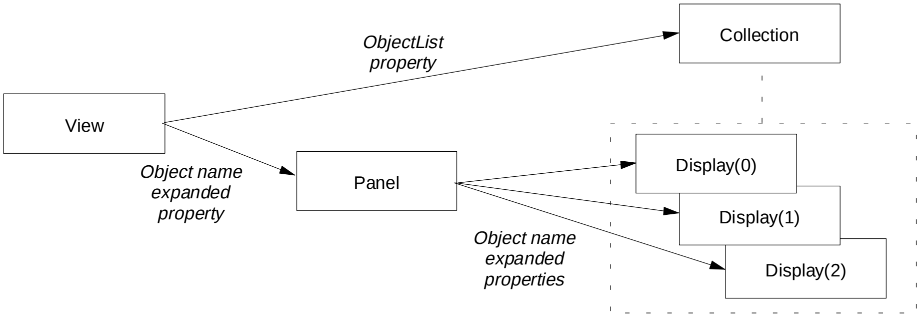
View and Panel classes as containers

The View class acts as a container for the Panel class. The View class is an abstract class used to create classes representing Approach forms, reports, and other views. The Panel class is also an abstract class. It is used to create classes representing header, footer, summary, body, and repeating panels.

The Panel class acts as a container for the Display class. The Display class is an abstract class used to create classes representing field boxes, text blocks, buttons, and other display elements.

The containment relationships between the View, Panel, and Display abstract classes are maintained between all of the classes derived from them. For example, a FieldBox object (derived from the Display class) is contained by a BodyPanel object (derived from the Panel class), which is contained by a Form object (derived from the View class).

These classes are related both through collections and expanded properties.



Using these containment relationships, you can identify a display element, such as a field box, in either of two ways:

- Using the name of the field box as an expanded property

  ```
  CurrentView.Body.fbxLastName
  ```

- Using the collection through the ObjectList property, where you know the index number corresponding to the field box

  ```
  CurrentView.ObjectList(0)
  ```

The BodyPanel object in the current view is identified through the expanded property Body.

Identifying containers using the Parent property

So far, this chapter has described containment relationships from the top down: from container to the contained object. Using the Parent property of most Approach objects, you can look at containment relationships in the opposite direction: if you know the contained object, the Parent property of that object identifies the container object. The container is the parent, and the contained object is the child.

When working with events, it is useful to be able to determine a parent object from a child object. An event receives the object that owns the event as the parameter Source. The Parent property allows you access to related objects using general references.

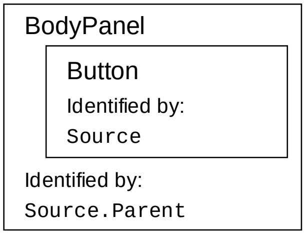
In the example at the beginning of the section "Approach object model," the Click event script for the button was specific to the object affected:

```
Sub Click(Source As Button, X As Long, Y As Long, _
   Flags As Long)

   Call CurrentDocument.Form1.Body.Background.Color.SetRGB _
      (COLOR_IRIS)

End Sub
```

Instead of identifying the SetRGB method relative to the CurrentDocument, you can write a more general script using what you know about the immediately affected objects. In this example, you know the following:

- In the Click event script, the variable Source identifies the Button object.
- The Button object and the object performing SetRGB (the Color object) are both contained by the BodyPanel object.
- The Parent property of the Button object identifies the BodyPanel object. That is, the parent of the Button object is the BodyPanel object.

Using the Button object as the starting point in the form of Source, you can then back your way up the hierarchy using the Parent property until you reach the shared object, BodyPanel.

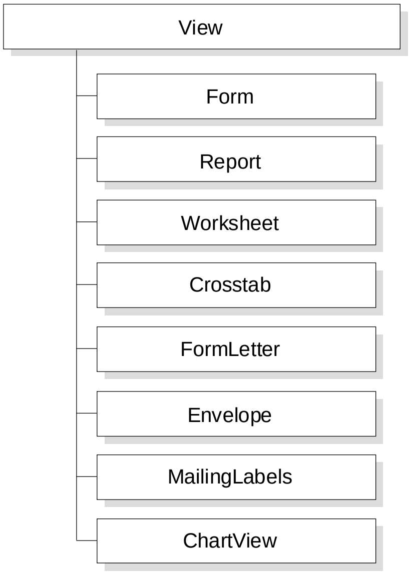


The Call statement in the Click event script becomes less specific and more easily applicable in other scripts:

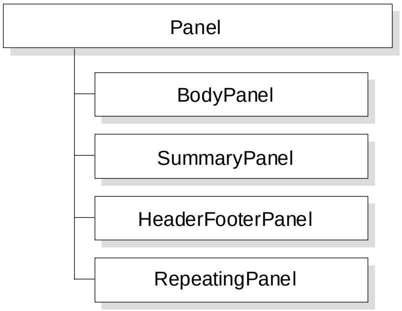
```
Sub Click(Source As Button, X As Long, Y As Long, _
   Flags As Long)

   Call Source.Parent.Background.Color.SetRGB(COLOR_IRIS)

End Sub
```

Approach inheritance relationships

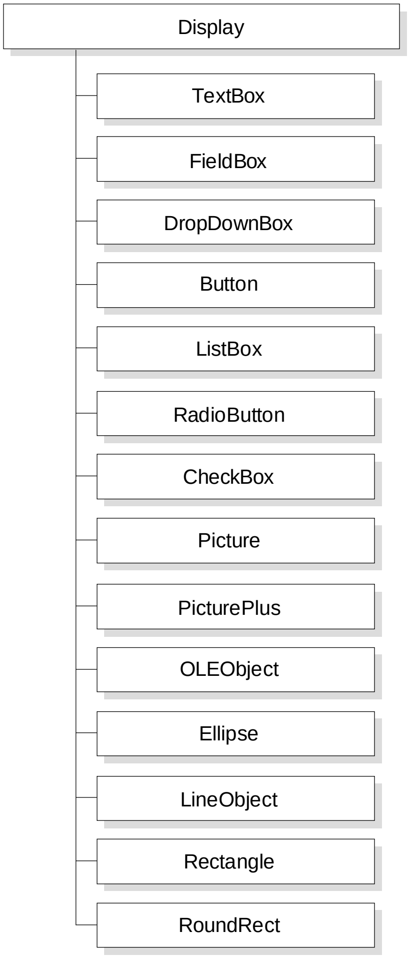
Approach provides abstract classes that exist only to create other derived classes. The following diagram shows the abstract class View and the classes derived from it.



The following diagram shows the abstract class Panel and the classes derived from it.



The following diagram shows the abstract class Display and the classes derived from it.



For more information about inheritance, see “Inheritance” in Chapter 2.

Approach predefined global product variables

Approach supports several predefined global product variables that greatly simplify specifying objects. Each of these variables lets you specify an object in the current Approach session, document, or view that was defined outside the current sub or function.

Variable	*Description*
CurrentApplication	Represents the current session of Approach and is the object at the top of the entire hierarchy.
CurrentDocument	Represents the current Approach document (.APR file) for the current session of Approach.
CurrentWindow	Represents the document window in the current Approach document. The DocWindow class includes properties and methods that control Approach interface elements, such as menus and sets of SmartIcons, that are independent of views or data.
CurrentView	Represents the current view (form, report, crosstab, and so on) in the current Approach document.

These variables replace all of the objects that are containers for the current object. For example, to specify an object contained by the current view, use the following syntax:

Correct:

CurrentView.*ObjName*

Do not use other variables to fill in the containment tree:

Incorrect:

CurrentApplication.CurrentDocument.CurrentView.*ObjName*

Another global product variable you will see in Approach LotusScript examples is Source. Source identifies the object for which an event occurs. For more information about using Source, see Chapter 4.

Creating new objects

To create a new object in LotusScript, you must declare a variable of the object's type and assign the new object to it. In Approach, there are two ways to create a new object using the LotusScript New keyword:

- Declare the object variable in one statement, and assign the new object to it in a second statement. For example, the following script creates a new Document object variable, MyDocument:

  ```
  Dim MyDocument As Document
  Set MyDocument = New Document(MyTable)
  ```

  The argument MyTable indicates the data source used to create the Document. In this case, the New keyword is treated as a method of the Document class. For information on the New method, search on "Methods" in the Approach Help Index.
- Declare the object variable and create the new object in the same statement. For example, the following statement creates a new Document object variable, MyDocument:

  ```
  Dim MyDocument As New Document(MyTable)
  ```

  In this case, the New keyword is an element of the Dim statement. For information on the Dim statement, search on "LotusScript" in the Approach Help Index, then click "LotusScript Index."

Approach classes

Approach classes give access to almost all of the functionality of Approach. There are 47 classes in Approach; the following sections describe the most important of these classes. Four of these classes are abstract classes, and the descriptions of these abstract classes give a general understanding of the classes that inherit from them. To explore the details of the classes derived from each abstract class, search on "Classes (LotusScript)" in the Approach Help Index.

The section "Approach containment hierarchy" earlier in this chapter describes how each class relates to other Approach classes.

Application class

The Application class represents the Approach application invoked from an executable. The Application class is not an abstract class.

Application properties

Application objects have the characteristics described by the following Application class properties:

Characteristics of the application	*Application class properties*
The .APR file, application window, and view currently in use.	ActiveDocument, ActiveDocWindow, ActiveView
The Approach application and window in use.	Application, ApplicationWindow
All documents, sets of SmartIcons, user language (such as English or French), menus, and windows in use in Approach. Documents and Windows both identify collections of objects contained in the Application object.	Documents, IconSets, Language, Menus, Windows
Full path and file name of the Approach session, only its name, or only its path.	FullName, Name, Path
The OLE Automation controller that invoked Approach.	Parent
The element that currently has focus; whether Approach is visible.	Selection, Visible

Application methods

Application objects can perform the operations described by the following Application class methods:

Actions that the application performs	*Application class methods*
Ending the session	CloseWindow
Making a Color object available	GetColorFromRGB
Opening an .APR file	OpenDocument
Running a sub or function	RunProcedure

Application events

An Application object responds to actions by the user or operating system as described by the following Application class events:

Actions that can affect the application	*Application class events*
Passing a value at the time Approach is invoked	Broadcast
Closing, creating, or opening an .APR file	DocumentClose, DocumentCreated, DocumentOpened
Checking, receiving, or sending mail from Approach	MailCheck, MailReceived, MailSend
Exiting Approach	Quit

Application class example

In the following example, the Application object identified by the predefined global product variable CurrentApplication determines the number and name of each .APR file that is currently open.

```
Sub ListDocuments

   Dim i As Integer
   Dim DocList As BaseCollection
   Dim DocCount As Integer

   ' Store the collection in DocList.
   Set DocList =  CurrentApplication.Documents

   ' Retrieve the number of documents open.
   DocCount = DocList.Count

   ' Print a summary of the number of documents open.
   Print ("There are " & DocCount & " documents open:")

   ' Print the index and name of each document.
   ' DocList starts counting at 0.
   For i = 0 To DocCount -1
      Print (i & "  " & DocList(i).Name)
   Next
End Sub
```

Window class

The Window class comprises the methods that describe the Approach window (ApplicationWindow class) and the .APR file window (DocWindow class). The Window class is an abstract class. You cannot create an instance of the Window class as such, but you can create and manipulate instances of the classes derived from it. The ApplicationWindow class and the DocWindow class inherit methods from the Window class.

Window properties

The Window class has no properties.

Some of the classes derived from the Window class have properties. For a list of the properties of each class, search on “Classes (LotusScript)” in the Approach Help Index, click “Approach classes,” and click a class name. In the topic, click the Class Members button.

Window methods

The Window class and all classes derived from it can perform the operations described by the following Window class methods:

Actions that windows perform	*Window class methods*
Changing the window display	Close, Maximize, Minimize, Restore
Retrieving operating system information about the window in order to call it from another product	GetHandle

Window events

The Window class has no events.

Some of the classes derived from the Window class do have events. For a list of the events of each class, search on “Classes (LotusScript)” in the Approach Help Index, click “Approach classes,” and click a class name. In the topic, click the Class Members button.

Window class example

Identify instances of classes derived from the Window class using the Window property of the container class. For example, to minimize the window for the current document, use the following statement:

```
Call CurrentDocument.Window.Minimize()
```

Minimize the window for the current application as follows:

```
Call CurrentApplication.Window.Minimize()
```

Document class

The Document class is not an abstract class. The Document class represents an application (an .APR file) running in Approach.

Note The term “application” refers to a Document object (an .APR file) that you are using or developing. The term “Application object,” however, refers to Approach itself rather than your application. In this chapter, “Approach” refers to an Application object, and “application” or “.APR file” refers to a Document object.

There are two ways to create a new Document object; they are described in “Creating new objects” earlier in this chapter.

Document properties

Document objects have the characteristics described by the following Document class properties:

Characteristics of documents	*Document class properties*
.APR file information entered by its creator	Author, Description, Keywords
.APR file information	FileName, FullName, Name, Path, User
The parent object of the Document object; that is, the Application object that contains the Document object	Parent
.APR file revision information	CreateDate, LastModified, Modified, NumRevisions
Connections to information used in the .APR file, including tables of calculated fields and variable fields	CalcTable, NumJoins, NumTables, Tables, VarTable
Information about the design of the .APR file	Menus, NamedFindSorts, NamedStyles, NumViews, Views, Window

Document methods

Document objects can perform the operations described by the following Application class methods:

Actions that documents perform	*Document class methods*
Creating a new Document object	New
Moving the application focus to a another application	Activate
Creating or deleting a calculated field	CreateCalcField, DeleteCalcField
Assigning a Table object to a variable	GetTableByName

Document events

The Document class has no events.

Document class example

This example uses the Document object for the Orders application and its Description property to print a description of the .APR file:

```
Sub DocDescription

   Print ("The Orders application: " & _
      CurrentApplication.Orders.Description)

End Sub
```

Table class

The Table class represents a table associated with an .APR file.

Note In this chapter, “table” refers to a single table in a database that may contain more than one table. Approach Help and other Approach documentation, however, use the term “database” broadly to mean both “database” and “table.”

You can identify a table through the Tables property of a Document object, which is a BaseCollection object representing all of the tables associated with the .APR file.

Table properties

Table objects have the characteristics described by the following Table class properties:

Characteristics of tables	*Table class properties*
Table information	FileName, FullName, Path, TableName
The Document object with which this Table is associated	Parent
Name of each field in a table	FieldNames
Counts of the fields and records in a table	NumFields, NumRecords

Table methods

Table objects can perform the operations described by the following Table class methods:

Actions that tables perform	*Table class methods*
Retrieving or replacing a table’s data source	CreateResultSet, ReplaceWithResultSet
Retrieving information about a field	GetFieldFormula, GetFieldOptions, GetFieldSize, GetFieldType

Table events

The Table class has no events.

Table class example

The following example uses the Table object variable MyTable to retrieve the name and data type for each field in the table.

```
Sub TableInfo

   Dim i As Integer
   Dim NumColumns As Integer
   Dim MyTable As Table
   Dim ColumnNames As Variant

   ' Store the first table object from the current document
   ' in MyTable.
   Set MyTable = CurrentDocument.Tables(0)

   ' Store the collection of field names in ColumnNames.
   ColumnNames = MyTable.FieldNames

   ' Print the field names and their data types.
   For i = 0 To MyTable.NumFields - 1

     Print (i & "  " & ColumnNames(i) & "   " & _
         Str$(MyTable.GetFieldType(ColumnNames(i))))
   Next

End Sub
```

View class

The View class comprises the properties and events that describe the views in Approach: forms, reports, worksheets, crosstabs, charts, form letters, mailing labels, and envelopes. The View class is an abstract class. You cannot create an instance of the View class as such, but you can create and manipulate instances of the classes derived from it.

The following classes are derived from the View class:

- Form
- Report
- Worksheet
- Crosstab
- ChartView
- FormLetter
- MailingLabels
- Envelope

View properties

All Approach views have characteristics described by the following View class properties:

Characteristics of views	*View class properties*
Appearance of the view	MenuBar, Visible
The application that the view appears in, the view name, and the view type	Document, Name, Type
The parent object of the view, that is, the Document object that contains the view	Parent
Data connection	MainTable
Macro executed by switching to or away from the view	OnSwitchFromMacro, OnSwitchToMacro

The classes derived from the View class have additional properties. These additional properties describe characteristics that make the various views different from one another. For example, the Report class includes the KeepRecsTogether property, but none of the other classes derived from the View class needs to disable page breaks within a record.

For a list of the properties of each class, search on “Classes (LotusScript)” in the Approach Help Index, click “Approach classes,” and click a class name. In the topic, click the Class Members button.

View methods

The View class has no methods.

Some of the classes derived from the View class do have methods. For a list of the methods of each class, search on “Classes (LotusScript)” in the Approach Help Index, click “Approach classes,” and click a class name. In the topic, click the Class Members button.

Each class derived from the View class has a New method for creating a new instance of the class. To create a new Form object, for example, you must declare a variable and assign the new Form object to the variable. For more information, see “Creating new objects” earlier in this chapter.

View events

The View class and all classes derived from it respond to actions by the user, application, or operating system as described by the following View class events:

Actions that affect views	*View class events*
Switching to or from another view	SwitchFrom, SwitchTo
Waiting the time indicated by a timer before executing or continuing an operation	UserTimer

View class example

Identify instances of classes derived from the View class using the name of the object as an expanded property of the Document object. For example, identify the Form1 view by using the following dot notation:

```
CurrentDocument.Form1
```

You can also identify the form by its location in the BaseCollection object contained through the Views property of the Document object. In this example, Form1 is the second item in the BaseCollection object:

```
CurrentDocument.Views(1)
```

The following example shows how to create a new Form object and assign a name to it:

```
Dim MyForm As Form
Set MyForm = New Form(CurrentDocument)
MyForm.Name = "My New Form"
```

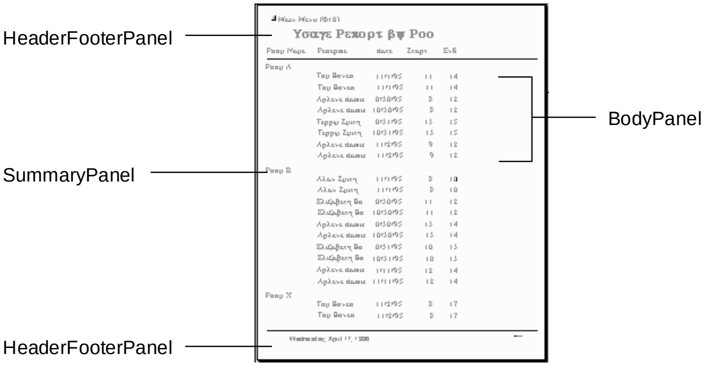
Panel class

The Panel class comprises the properties and methods that describe the common characteristics and operations of the following components of Approach views:

- The body panel of a form, report, worksheet, crosstab, chart, form letter, mailing label, or envelope
- The header, footer, and summary panels of a report
- The repeating panel on a form

The Panel class is an abstract class. You cannot create an instance of the Panel class as such, but you can create and manipulate instances of the classes derived from it.

The following illustration shows the classes derived from the Panel class and the panels in a report you can create by using these classes. The RepeatingPanel is not shown, but is also derived from the Panel class.



You cannot name objects representing panels in Approach because they have no Name property. To identify a panel in a script, use the following identifiers created by Approach for the panel:

Panel type	*Identifier*
BodyPanel	Body
SummaryPanel	Summary, Summary1, Summary2, and so on
HeaderFooterPanel	Header, Footer
RepeatingPanel	RepeatingPanel, RepeatingPanel1, RepeatingPanel2, and so on

Panel properties

All Approach panels have characteristics described by the following Panel class properties:

Characteristics of panels	*Panel class properties*
Size of the panel	Height
Appearance of the panel	Background, Border, NamedStyle
The parent object of the Panel object, that is, the view in which the panel resides	Parent
The type of panel	Type

The classes derived from the Panel class have additional properties. These additional properties describe characteristics that make the various panels different from one another. For example, the SummaryPanel class includes the property PageBreak, which indicates that a page break occurs each time the panel appears. None of the other panels has this property.

For a list of the properties of each class, search on “Classes (LotusScript)” in the Approach Help Index, select “Approach classes,” and select a class name. In the topic, click the Class Members button.

Panel method

The Panel class and all classes derived from it can perform the operation described by the following Panel class method:

Action that panels perform	*Panel class method*
Creating a named style from the display element attributes	MakeNamedStyle

Panel events

The Panel class has no events.

Some of the classes derived from the Panel class do have events. For a list of the events of each class, search on “Classes (LotusScript)” in the Approach Help Index, click “Approach classes,” and click a class name. In the topic, click the Class Members button.

Panel class example

Identify instances of classes derived from the Panel class using the name of the object as an expanded property. For example, to identify the second summary panel placed in a report, use the following dot notation:

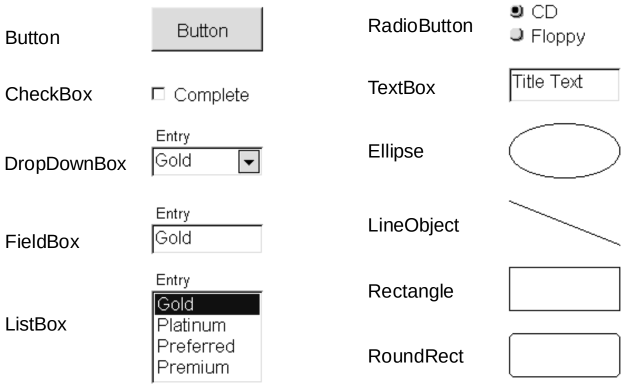
```
CurrentView.Summary1
```

In an event script for an object in the same view, identify the summary panel as follows:

```
Source.Summary1
```

Display class

The Display class comprises properties, methods, and events that describe the most common elements in Approach: text blocks, field boxes, buttons, pictures, drawn objects, and so on. The Display class is an abstract class. You cannot create an instance of the Display class as such, but you can create and manipulate instances of the classes derived from it. The following illustration shows the classes derived from the Display class and the display elements you can create by using these classes:



The OLEObject, Picture, and PicturePlus classes are not shown, but they are also classes derived from the Display class.

Display properties

All Approach display elements have characteristics described by the following Display class properties:

Characteristics of display elements	*Display class properties*
Location and size of the element in the view	Top, Left, Height, Width, Page
The element name, parent object, and type	Name, Parent, Type
Position in tab order	TabOrder, TabStop
The macro executed by tabbing to or away from the element	MacroTabIn, MacroTabOut
Print behavior	SlideLeft, SlideUp, NonPrinting, ShowInPreview, Visible

Some of the classes derived from the Display class have additional properties. These additional properties describe characteristics that make the various display elements different from one another. For example, the FieldBox class has the DataField and DataTable properties to indicate the connection between a field in a table and the field box in a view. Text blocks do not have these properties because they do not display data from a table.

For a list of the properties of each class, search on “Classes (LotusScript)” in the Approach Help Index, click “Approach classes,” and click a class name. In the topic, click the Class Members button.

Display methods

The Display class and all classes derived from it can perform the operations described by the following Display class methods:

Actions that display elements perform	*Display class methods*
Changing the arrangement of elements in the view	BringToFront, SendToBack
Moving the application focus to the element	SetFocus
Updating the data displayed by the element	Refresh
Placing a new element in a view	InsertAfter
Creating a named style from the display element attributes	MakeNamedStyle

All classes derived from the Display class also have the New method for creating an instance of the class. The New method is different for each derived class, so it isn’t part of the Display class description. To create a new FieldBox object, for example, you must declare a variable and assign the new FieldBox object to it. For more information, see “Creating new objects” earlier in this chapter.

Some of the classes derived from the Display class have additional methods. These additional methods describe operations that make the various display elements behave differently from one another. For example, the RadioButton and CheckBox classes include the SetState method to predefine the value displayed by the field. Other display elements do not have this method.

For a list of the methods of each class, search on “Classes (LotusScript)” in the Approach Help Index, select “Approach classes,” and select a class name. In the topic, click the Class Members button.

Display events

The Display class and all classes derived from it respond to actions by the user, application, or operating system as described by the following Display class events:

Actions that affect display elements	*Display class events*
Clicking or double-clicking the element	Click, DoubleClick
Selecting an element with the mouse to drag it and releasing the mouse button	MouseDown, MouseUp

Display class example

Identify instances of classes derived from the Display class using the name of the object. For example, to identify a field box that displays last name information, use the following dot notation:

```
CurrentView.Body.fbxLastName
```

In an event script for an object on the same panel as the field box, identify the field box as follows:

```
Source.Parent.fbxLastName
```

Find class

The Find class represents all of the information needed to create a found set. It automates the process of creating a find request and running the find. After you create a Find object, run the find using the FindSort method of the DocWindow class.

There are two ways to create a new Find object; they are described in “Creating new objects” earlier in this chapter.

For more information about creating and running a find, see the examples described in “Top tasks” later in this chapter.

The following classes are similar to the Find class:

- Sort class

  Sorts records. Run a sort using the FindSort method of a DocWindow object. You can run a sort at the same time as a find.

- FindDistinct class

  Finds records with unique field values. You can specify a FindDistinct object in any operation where you can use a Find object.

- FindDuplicate class

  Finds records with duplicate field values. You can specify a FindDuplicate object in any operation where you can use a Find object.

Find properties

Find objects have the characteristics described by the following Find class property:

Characteristic of finds	*Find object property*
The FindDuplicate or FindDistinct object contained by the Find object	FindSpecial

Find methods

Find objects can perform the operations described by the following Find class methods:

Actions that finds perform	*Find class methods*
Adding an AND or OR condition to the find	And, Or
Retrieving find condition details and the number of conditions in the find	GetAt, GetCount
Clearing conditions from the find	RemoveAll
Creating a find	New

Find events

The Find class has no events.

Find class example

The following example shows how to create the Find object FindLast, and how to run the find.

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

   ' Declare a DocWindow object variable.
   Dim MyDocWin As DocWindow

   ' Retrieve the active DocWindow.
   Set MyDocWin = CurrentApplication.ActiveDocWindow

   ' Declare a Table object variable.
   Dim  MyTable As String

   ' Retrieve the name of the main table for the view.
   MyTable = CurrentDocument.Tables(0).TableName

   ' Create a new Find object to search for a last name.
   Dim FindLast As New Find (MyTable & ".Last", "Garcia")
   ' Find last name.
   Call MyDocWin.FindSort (FindLast)

End Sub
```

Connection class

Using LotusScript in Approach, you can bypass the user interface and access data directly from a table. The table can be one already associated with the .APR file, or a distinct table. This batch process involves three classes:

- The Connection class represents all of the information needed to make a connection to a specific database format.
- The Query class represents the selection information for which records from a table are returned in the batch operation.
- The ResultSet class represents the records returned from the connection to the table.

Accessing data through this batch process can be very fast for the following operations:

- Examining or modifying many or all records in a very large database
- Performing calculations using data from a large number of records

For illustrations of the relationships between the Table, Connection, Query, and ResultSet objects, see the examples described in “Top tasks” later in this chapter.

There are two ways to create a new Connection object; they are described in “Creating new objects” earlier in this chapter.

Accessing Approach data from other Lotus products

The Connection, Query, and ResultSet objects can be made available to other Lotus products that use LotusScript, so that applications written in those products can access Approach data. These objects are exposed to other Lotus products via a LotusScript Extension (LSX) module, DBENGN01.LSX, which combines functionality of the Connection, Query, and ResultSet objects. An LSX module is like a Dynamic-Link Library (DLL) that gives you access to the data.

For example, to access Approach data through a Word Pro application, do the following:

1. Click the Object drop-down box in the IDE.
2. Select !Globals from the list.

3. Type the following statement in the Script Editor:

```
UseLSX "<drive>\<path>\dbengn01.lsx"
```

drive refers to the drive on which the Approach DBENGN01.LSX file is stored.

path refers to the directory path where the Approach DBENGN01.LSX file is stored.

4. Write scripts in Word Pro using the Approach Connection, Query, and ResultSet objects to retrieve data from a table.

Connection properties

Connection objects have the characteristics described by the following Connection class properties:

Characteristics of connections	*Connection class properties*
Record commit status	AutoCommit
Information about the PowerKey and user information needed to make the data connection	DataSourceName, Password, UserId
Status of the connection	IsConnected

Connection methods

Connection objects can perform the operations described by the following Connection class methods:

Actions that connections perform	*Connection class methods*
Specifying the table type to access	ConnectTo
Retrieving error information	GetError, GetErrorMessage, GetExtendedErrorMessage
Retrieving information about available data format types and tables	ListDataSources, ListFields, ListTables
Committing or rolling back record changes	Transactions
Deleting and creating the connection	Disconnect, New

Connection events

The Connection class has no events.

Connection class example

The following example shows how a new Connection object is created and set to a data source type. See the example in “ResultSet class” to understand more of the steps involved in creating a data connection.

```
Dim MyConnection as New Connection
Call MyConnection.ConnectTo("ODBC Data Sources")
```

Query class

The Query class represents an SQL (Standard Query Language) statement. This statement is used to search through a table and return records that match given find conditions. Use a Query object with a Connection object and a ResultSet object for data access operations.

For illustrations of the relationships between the Table, Connection, Query, and ResultSet objects, see the examples described in “Top tasks” later in this chapter.

There are two ways to create a new Query object; they are described in “Creating new objects” earlier in this chapter.

Query properties

Query objects have the characteristics described by the following Query class properties:

Characteristics of queries	*Query class properties*
Connection to data source for the query	Connection
Find conditions specified using SQL and the table to return if there are no find conditions	SQL, TableName

Query methods

Query objects can perform the operations described by the following Query class methods:

Actions that queries perform	*Query class methods*
Retrieving query error information	GetError, GetErrorMessage, GetExtendedErrorMessage
Creating a new query object	New

Query events

The Query class has no events.

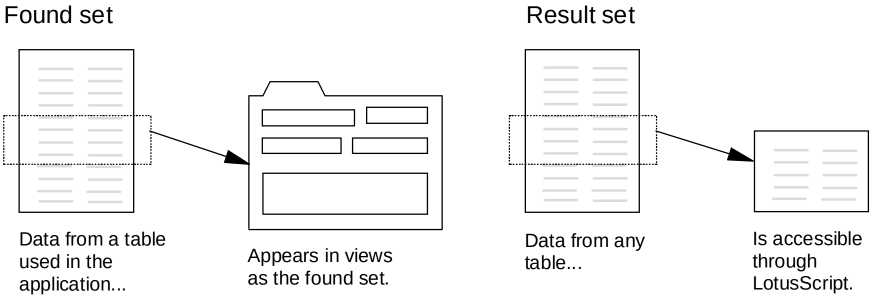
Query class example

The following example shows how to create a Query object to retrieve all records from a table.

```
Sub RunQuery

   Dim MyConnection As New Connection
   Dim ThisQuery As New Query
   Dim ThisTable As String
   Dim ThisResultSet As New ResultSet

   ' Open the connection.
   Call MyConnection.ConnectTo ("dBASE IV")

   ' Store the name of the table to find records in.
   ' In this case, it is the Orders table.
   ThisTable = "Orders.dbf"

   ' Use this connection and table in the query definition.
   Set ThisQuery.Connection = MyConnection

   ThisQuery.TableName = CurrentDocument.Tables(0).Path & _
      ThisTable

   ' The definition and execution of a result set must
   ' follow here.

End Sub
```

ResultSet class

The ResultSet class represents all of the information returned from a query. A result set is exactly like a found set, except that the data from a result set does not appear in the user interface. Instead, the result set data is available as a table that you can access through LotusScript.


Found set
Result set
Data from a table used in the application...
Appears in views as the found set.
Data from any table...
Is accessible through LotusScript.

Use a ResultSet object with a Query object and a Connection object for data access operations.

For illustrations of the relationships between the Table, Connection, Query, and ResultSet objects, see the examples described in “Top tasks” later in this chapter.

There are two ways to create a new ResultSet object; they are described in “Creating new objects” earlier in this chapter.

ResultSet properties

ResultSet objects have the characteristics described by the following ResultSet class properties:

Characteristics of result sets	*ResultSet class properties*
The record currently in use	CurrentRow
Flags that indicate the first and last records in the result set	IsBeginOfData, IsEndOfData
The Query object used to create the result set	Query
Flags that indicate the status of the result set	IsReadOnly, IsResultSetAvailable

ResultSet methods

ResultSet objects can perform the operations described by the following ResultSet class methods:

Actions that result sets perform	*ResultSet class methods*
Adding or deleting a record	AddRow, DeleteRow
Moving between records	FirstRow, LastRow, NextRow, PreviousRow
Closing the result set	Close
Creating a new result set	New
Setting how records are updated	Options
Filling the result set with data from a table	Execute
Writing record changes to a table	UpdateRow
Retrieving error information	GetError, GetErrorMessage, GetExtendedErrorMessage
Retrieving and setting elements of the find conditions	GetParameter, GetParameterName, GetValue, SetParameter, SetValue
Retrieving and setting field information	FieldExpectedDataType, FieldId, FieldName, FieldNativeDataType, FieldSize
Retrieving statistics about the result set	NumColumns, NumParameters, NumRows

ResultSet events

The ResultSet class has no events.

ResultSet class example

The following example shows how the ResultSet object ThisResultSet is given a query and executed.

```
Sub RunQuery

   Dim MyConnection As New Connection
   Dim ThisQuery As New Query
   Dim ThisTable As String
   Dim ThisResultSet As New ResultSet

   ' Open the connection.
   Call MyConnection.ConnectTo ("dBASE IV")

   ' Store the name of the table to find records in.
   ' In this case, it is the first table for the document.
   ThisTable = CurrentDocument.Tables(0).TableName

   ' Use this connection and table in the query definition.
   Set ThisQuery.Connection = MyConnection
   ThisQuery.TableName = CurrentDocument.Tables(0).Path & _
      ThisTable

   ' Use this query in the result set definition.
   Set ThisResultSet.Query = ThisQuery

   ' Create the result set.
   Call ThisResultSet.Execute()

   ' Now the ResultSet is available to use.

End Sub
```

Recording scripts in Approach

Approach can record a transcript of actions you perform in Approach. Store the transcript as a script in an .LSS file or in the IDE at the insertion point; or as a macro. The resulting script or macro can be edited or copied.

Recording Approach tasks as LotusScript involves the following operations:

- Practicing the actions you intend to record.
- Determining how to store the transcript: Choose Edit - Record Transcript.
- Performing the actions you want to record.
- Stopping the recording process: Choose Edit - Stop Recording.

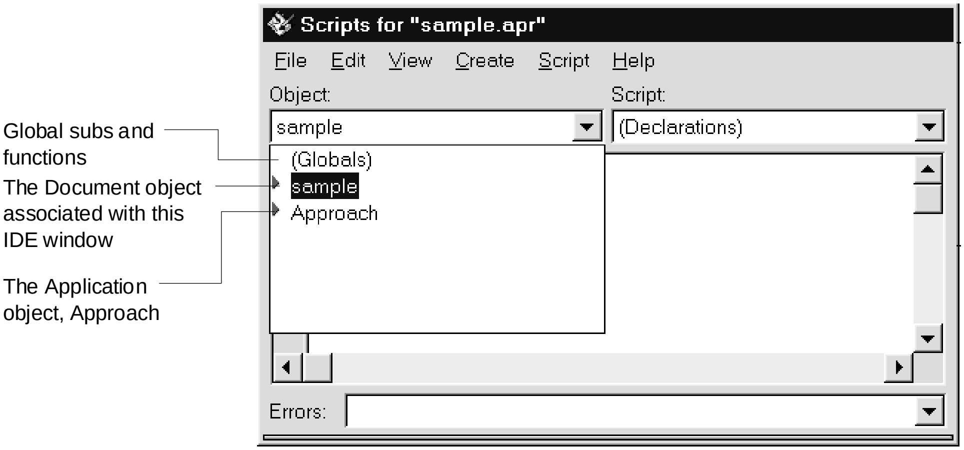
Using the IDE in Approach

This section describes Approach-specific behavior of the IDE. For general IDE operation, see Chapter 3.

In Approach, open the IDE by choosing Edit - Show Script Editor.

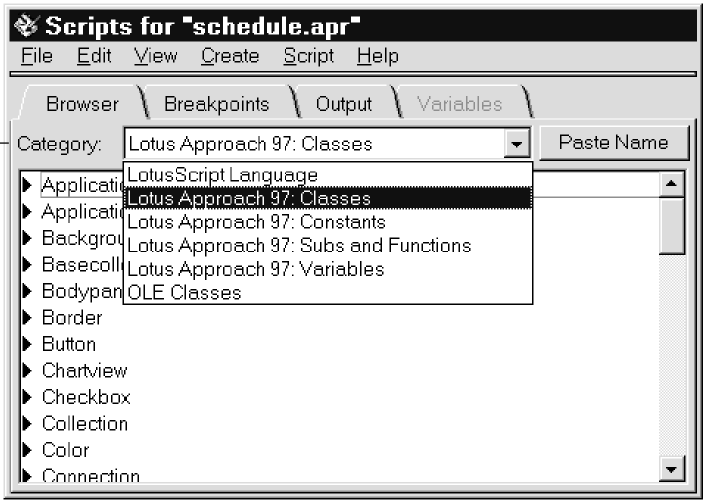
The Object drop-down box in the IDE lists the Approach objects in their containment structure. When you first open the list, you see the following items:

- Globals, which lists the global subs and functions
- The name of the Document object (.APR file) associated with this IDE window
- Approach, which is the default name of the Application object



When you select the Document object, you see a list of the views it contains. Each view opens to show the objects it contains, and so on through the containment hierarchy.

The Browser lists Approach classes, constants, subs and functions, and variables. Detailed information is available for each entry in the Browser by pressing **F1**.


Scripts for "schedule.apr"
File Edit View Create Script Help
Browser
Breakpoints
Output
Variables
Category:
Lotus Approach 97: Classes
Paste Name
LotusScript Language
Lotus Approach 97: Classes
Lotus Approach 97: Constants
Lotus Approach 97: Subs and Functions
Lotus Approach 97: Variables
OLE Classes
Applicati
Applicati
Backgrou
Basecoll
Bodypan
Border
Button
Chartview
Checkbox
Collection
Color
Connection

Select a category to see lists of relevant LotusScript elements.

Script templates

As you use LotusScript, you may find yourself entering the same function or other code in numerous scripts. In the Script Editor, Approach provides several frequently used scripts as script templates. You can use these templates to insert script statements into the current script. You can also add your own scripts to the list of templates available through the Script Editor.

Inserting a script template

To insert a script template:

1. In the Script Editor, open the script to which you want to add the script template.
2. Place the insertion point in the script where you want to insert the template.
3. Choose Script - Insert Template.
4. Select a script template.

Template name	*Description*
Automation example	Finds field values less than a given number and stores them statically in a new Word Pro document.
Connection example	Connects to a sample DB2 database.
Create calculated field	Creates a calculated field for the current .APR file.
Notes connect	Connects to a sample Notes database.

5. Click OK.

Approach places the script and code comments in the current script at the insertion point. You can and will probably want to modify the code that was inserted from the script template because the templates contain example text and variable names.

Creating a script template

You can add scripts to the list of templates available through the Script Editor. To create a template:

1. In the Script Editor, select the script statements from which you want create a template.
2. Choose Edit - Copy.
3. Paste the script in a text editor outside Approach.
4. Add a comment line to the top of the script.

   For example, if your template contains statements to add today’s date to a field, enter the following comment to the top of the file:

   ```
   ' Insert today's date
   ```

   The comment text appears in the Insert Script Template dialog box.
5. Save the text file (with a .TPL extension) to the directory C:\<*PATH*>\APPROACH\SCRIPTS.

   PATH is either the default SmartSuite directory or an alternative directory selected during Install.

Customizing the Approach user interface

Use LotusScript in Approach to enhance the usability of your application by customizing the user interface. You can use LotusScript or macros to build enhancements, such as the following:

- Add buttons and menu items to execute tasks that you define.
- Execute tasks based on data the user enters in a view.
- Create dialog boxes to control how users enter information.

The following sections describe how to associate a script with a button or menu in the application.

Attaching a script to a button

Approach makes it easy for you to attach a script (or a macro) to a button in a view. If the button isn't already in the view, add the button by clicking the Button icon on the Tools palette.

To name objects you create in Design

Approach assigns a name to all elements you create in a view. For example, if you add a button to a form, Approach automatically names the button ObjButton. You can see the name in the InfoBox and change it there.

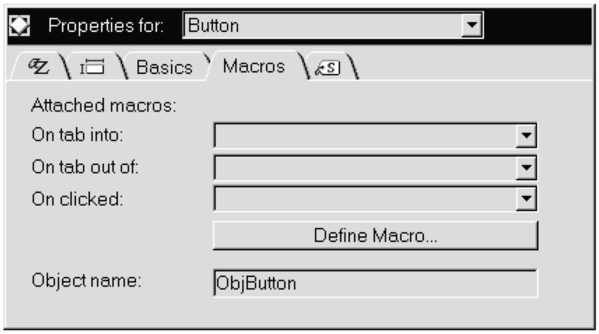
To change the name of a button or any other display element that you create in Design:

1. In Design, double-click the button.

2. Click the Macros tab in the InfoBox.

The button's name appears in the "Object name" box.

3. Enter the new name.



Properties for: Button

Object name: ObjButton

To attach a script to a button

1. Choose Edit - Show Script Editor.

2. Select the Button object by its name in the Object drop-down box.

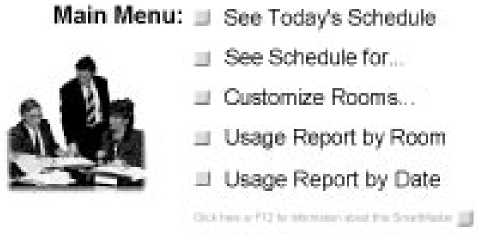
To find the Button object, you have to open each object in the containment hierarchy till you reach the container that contains the button.

3. Verify that Click appears in the Script drop-down box.

The Script Editor shows an empty Sub statement for the Click event of the button. The script you enter between the lines Sub and End Sub runs when the user clicks the button.

4. Enter the script.

The following illustration shows an Approach view that offers a list of tasks a user can do in a room-reservation application. The button to the left of each menu item has a script (or macro) attached to it. When the user clicks a button, the script or macro runs.



For example, the script attached to the "See Schedule for" button looks like this:

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

   Set CurrentWindow.ActiveView = CurrentDocument.Enter~ Date

End Sub
```

The Enter Date view appears as a dialog box in which the user indicates which day of reservations to display. The space in the view name is represented as a tilde (~) followed by a space.

Attaching a script to a menu item

Approach lets you attach a script to a custom menu item so a user can run the script by selecting the menu item. The script must be a global sub or function.

Attaching a script to a menu item requires you to do the following:

- Create a global sub or function in the IDE.

  For information on how to create a sub or function, see "Creating scripts" in Chapter 3.

- Create a macro that runs the sub or function.

  For information on how to create a macro, search on "Macros, creating" in the Approach Help Index.

- Choose the macro as the Item Action when creating the menu item.

  For information on how to create a custom menu item, search on "Menus, creating" in the Approach Help Index.

Attaching a script to a function key

Approach lets you attach a script to a function key so a user can run the script by pressing the key. The script must be a global sub or function.

Attaching a script to a function key requires you to do the following:

- Create a global sub or function in the IDE.

  For information on how to create a sub or function, search on "LotusScript" in the Approach Help Index, click "LotusScript Index," and then search on "Subs, creating."
- Create a macro that runs the sub or function.

  For information on how to create a macro, search on "Macros, creating" in the Approach Help Index.
- Specify the function key in the macro definition.

Top tasks

This section describes some common Approach tasks and gives LotusScript solutions for them. The code for these examples is available in the sample files directory. File names are listed with the individual examples.

The following tasks are described in this section:

- Switching between views in a document
- Accessing data from a database using a batch process
- Accessing data from a Notes database
- Finding records using the Find object
- Modifying records
- Displaying data from a result set in a view
- Creating a document to display the result set
- Controlling how users enter data
- Changing the summaries in a report
- Inserting and using OLE controls

Switching between views in a document

One of the most common ways to control the flow of an application is to create a view for each task that a user may perform in the application. Provide a menu view to guide users through the tasks. For each task, attach a script (or a macro) to a button in the menu view that switches to a view that you have set up for the task.

You can easily automate switching views with a macro. If switching views is only one out of a series of actions being automated, a script may be more suitable.

The following example comes from the Approach Meeting Room Scheduler SmartMaster™ application. To open this SmartMaster, choose File - New Database and select the Meeting Room Scheduler. The script appears in the Click event for the btnToday object on the Start view. When the user clicks this button, the script is executed and the following occurs:

- The application switches to the Schedule Display view in order to show the rooms and times that are reserved for a specific date.
- The current date (from the system) is formatted and appears in a text block at the top of the view.
- Schedule information in the view from previous uses is cleared.
- Today's schedule information appears in the view.

The following scripts accomplish this sequence of operations:

This script appears in the Click event of the btnToday object on the Start view.

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

   ' Display the schedule for the current system date.
   Call DisplaySchedule(Format$(Now, "m/d/yy"))

End Sub
```

The function DisplaySchedule is a global function in the same .APR file. The tilde (~) precedes a space in the name of a view.

```
Function DisplaySchedule(DateToDisplay As String)

   ' DateToDisplay–Schedule date to display information for.

   ' Change to the Schedule Display view.
   Set CurrentWindow.ActiveView = _
      CurrentDocument.Schedule~ Display

   ' Display the passed-in date in fbxDateDisplay field box.
   CurrentView.Body.fbxDateDisplay.Text = DateToDisplay

   ' Clear schedule information from the view.
   Call ClearDisplay()             ' Global sub

   ' Fill the schedule information for the date.
   Call ReadBlock(DateToDisplay)   ' Global function

End Function
```

Accessing data from a database using a batch process

There are two strategies for working with data using LotusScript in Approach:

- You can access values in records through the fields in a view.
- You can bypass the user interface and access data directly from a database.

LotusScript lets you automate the first type of data access by controlling elements of the Approach interface that you are already familiar with: placing fields in a view and modifying the value entered in the field.

The second strategy accesses the data in a fast batch process without using the Approach user interface. This data access process involves the following operations:

- Establishing a link to a database using a Connection object
- Selecting the records to access from the database using a Query object
- Storing the required information locally while it is in use using a ResultSet object

To understand more about when to use this method for accessing data, see “Connection class” earlier in this chapter.

The following example illustrates each of the steps for accessing data directly. It uses these objects:

- Document (referred to using the global product variable CurrentDocument)
- Connection
- Query
- ResultSet

The following script comes from the Approach Meeting Room Scheduler SmartMaster application. To open this SmartMaster, choose File - New Database and select the Meeting Room Scheduler. The script is a global sub.

```
Sub ReadBlock(DateReserved As String)

   ' DateReserved      Date formatted as a string

   ' Declare objects for connecting to the
   ' reservation database.

   Dim C As New Connection
   Dim Q As New Query
   Dim RS As New ResultSet

   Dim s As Double         ' Start time of existing
                           ' reservation
   Dim f As Double         ' End time of existing reservation
   Dim Row As String       ' Room reserved
   Dim n As String         ' Reservation owner
   Dim Tname As String     ' A shorter reservation table name
                           ' reference
   Dim FullTname As String ' Table name and path reference

   ' Collect the name of the first table associated with the
   ' document, numbered starting at 0.
   Tname = CurrentDocument.Tables(0).TableName

   ' Place the names of the current rooms in the view.
   Call DisplayRooms()

   ' Build the connection to retrieve the reservation
   ' information for the passed-in date. This is a standard
   ' data-access sequence. To reuse it, modify SQL SELECT
   ' statement as needed.

   ' Note that the database is dBASE IV in this case.
   If C.ConnectTo("dBASE IV") Then

      Set Q.Connection = C

      ' Table name for the query needs to have full path.
      FullTname =  CurrentDocument.Tables(0).Path & Tname & _
         ".dbf"

      ' The query is set to retrieve values from all the
      ' table fields for records whose Date Reserved field
      ' matches the date.
      ' Note that the syntax for the SQL SELECT statement
      ' requires extra quotation marks to define
      ' the string oncatenation.

      Q.SQL = "SELECT * FROM """+FullTname+""""+ Tname+ _
         " WHERE ((" + Tname+".""Date Reserved"" = ' " _
         +DateReserved+"' ))"

      ' Assign this query to the result set.
      Set RS.Query = Q

      ' Use the result set to fill in the reservation
      ' information in the view.
      ' If the result set was created successfully, then...
      If (RS.Execute) Then

         ' Confirm that there are reservations for this date.
         If (RS.NumRows) Then

            RS.FirstRow    ' Go to the first record in the
                           ' result set.

            ' Loop through all of the records in the result
            ' set and display the reservation information
            ' in the view.

            Do
               s = RS.GetValue("Start Time")
               f = RS.GetValue("End Time")
               Row = RS.GetValue("Room Name/Number")
               n = RS.GetValue("Reserved By")

               ' Build a text block in the view with the
               ' reservation information from this pass
               ' through the loop by calling the global
               ' function DisplayBlock.

               Call DisplayBlock(n, s, f, row)

            Loop   While RS.NextRow
         End If   ' NumRows not 0
      End If      ' Result set successful
   End If         ' Connection successful

   ' Close the connection to allow other connections
   ' to this database.
   C.Disconnect
End Sub
```

Accessing data from a Notes database

The Connection, Query, and ResultSet objects let you access Notes data from Approach. You might want to do this to create Approach reports or to search for data in a Notes database.

To create a connection to a Notes database, you must know the following:

- The data-source type of the database: Is the database on a server, on your local hard disk, or on the workspace?
- The name of the table to search.
- The user name and password, if required to access the table.
- The fields that you want to use in the find.

Connecting to a Notes database on a server involves the following operations:

- Determining the exact server name. To do so, choose File - Open; from "Files of type" select Lotus Notes - Server; note the name of the server you are working with. The following is an example of a server name:

  ```
  CN=Approach_OU=SJC_OU=A_O=Lotus
  ```

- Changing the underscores (_) in the server name to forward slashes (/). The example server name becomes the following:

  ```
  CN=Approach/OU=SJC/OU=A/O=Lotus
  ```

- Determining the database file name, including path. An example database file name is EXAMPLES\BUSCARD.NSF.
- Determining the user name and password, if necessary to access the database name.
- Building ConnectTo method arguments. The following table describes these arguments.

ConnectTo method argument	*Value*
DataSourceType	Lotus Notes - Server
UserId	A string
Password	A string
Database	ServerName!DatabaseName

The following connection script shows how these pieces come together in the ConnectTo method statement:

```
Dim Con As Connection
Dim Qry As Query

' Make the connection.
' No UserId or Password is required to access this database.
If Con.ConnectTo("Lotus Notes - Server",,, _
 "CN=Approach/OU=SJC/OU=A/O=Lotus!Examples\Buscard.nsf") then

   Set Qry.Connection = Con

   ' Continue building query and result set.
   Con.Disconnect

End If
```

The following example illustrates a connection to a local Notes database. The text of this script is stored in DW08_S1.LSS in the sample files directory. It does the following:

- Determines the types of connections you can make and prints a list to the Output panel of the IDE.
- Makes a “Lotus Notes - Local” data source connection and connects to a database file called NAMES.NSF.
- Lists all of the tables in NAMES.NSF and opens one called People.
- Searches that table in the First Name and Last Name fields and prints the names of the people in that table to the IDE.

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

   ' Declare objects for a Connection, Query, and ResultSet.
   Dim MyConnection As New Connection
   Dim MyQuery As New Query
   Dim MyResultSet As New ResultSet

' Declare a set of variables for an array, and for the
' arguments of the above objects.
Dim MyArray As Variant            ' Array for temporary
                                  ' storage
Dim MyPath As String              ' Data source path
Dim MyDatabaseSource As String    ' Current data source type
Dim MyDatabase As String          ' Database name
Dim MyTable As String             ' Table name
Dim MyUserId As String            ' User login name
Dim MyPassword As String          ' User password
Dim i As Integer                  ' MyArray index
Dim LastName As String            ' Data retrieved from table
Dim FirstName As String           ' Data retrieved from table

' Print a list of data source types available in the Output
' panel of the IDE. Use this list to determine the exact
' string required (in the ConnectTo method) to make the
' connection to the data source.
MyArray = MyConnection.ListDataSources()

' Use LotusScript functions LBound (lower bound) and UBound
' (upper bound) to determine the limits of the data source
' types array.
For i = LBound(MyArray) To UBound(MyArray)

   ' Print the data source types to the Output panel of
   ' the IDE.
   Print "Data source("i") = "MyArray(i)
Next

' Set the arguments for the Connection object.
' Tip: You could use an input box to get this information
' from the user and connect to whatever the user specifies.
MyDatabaseSource = "Lotus Notes - Local"

' Change this path to match your operating system.
MyPath = "C:\NOTES\DATA\"
MyDatabase = "NAMES.NSF"

' This table has no user or password requirements, so these
' strings are blank.
MyUserId = ""
MyPassword = ""

' Connect to the database.
If MyConnection.ConnectTo(MyDatabaseSource, MyUserId, _
   MyPassword, MyPath & MyDatabase) Then

   ' List the available tables in the database.
   MyArray = MyConnection.ListTables()

   For i = LBound(MyArray) To UBound(MyArray)

      ' Print the data source types to the IDE Output panel.
      Print "Table("i") = "MyArray(i)
   Next

   ' Specify the table that data will be extracted from.
   MyTable = "People"
```

```
  ' Set the Connection property of the Query object to
  ' the current connection.
  Set MyQuery.Connection = MyConnection

  ' Specify the table to be searched, including path.
  ' The ampersand (&) concatenates pieces of the string.
  MyQuery.TableName = MyPath & MyDatabase & "\" & MyTable

  ' Set the Query property of the ResultSet to the query.
  Set MyResultSet.Query = MyQuery

 ' Get the data from the table and put it in the result set.
If (MyResultSet.Execute) Then

     ' Make sure there is data in the table.
     If (MyResultSet.NumRows) Then

        ' Start at the first row.
        MyResultSet.FirstRow

        ' Loop through the records and get the values from
        ' the First_Name and Last_Name fields.
        Do
           LastName = MyResultSet.GetValue("Last_Name")
           FirstName = MyResultSet.GetValue("First_Name")

           ' Print the data to the Output panel of the IDE.
           Print "Person" & MyResultSet.CurrentRow & _
              "in the table is " & FirstName & _
              " " & LastName

        ' Continue looping until there are no more rows.
        Loop   While MyResultSet.NextRow
     Else

        ' If the table is empty, warn the user.
        MessageBox "The table is empty.", MB_OK + _
           MB_ICONINFORMATION + MB_APPLMODAL, "Empty Table"

     End If   ' If there is data in the table.
  Else

     ' If the result set was not successful, warn the user.
     MessageBox "The query did not succeed. Check the " & _
        "connection.", MB_OK + MB_ICONINFORMATION + _
        MB_APPLMODAL, "Unsuccessful Query"

  End If   ' If the result set was successful.
```

```
Call MyConnection.Disconnect()
Else

   ' If the connection was not successful, warn the user.
   MessageBox "Connection failed", MB_OK + _
      MB_ICONINFORMATION + MB_APPLMODAL, "Connection"

End If   ' If the connection was successful.

End Sub
```

Finding records using the Find object

There are two ways to automate finding records:

- Create a found set from records in a main or detail table associated with the specified .APR file. Specify the first find condition with the New method, and add more find conditions using the And or Or method of the Find object.
- Create a result set by accessing data from any table through Connection, Query, and ResultSet objects. Specify find conditions using the SQL property of the Query object. The retrieved data is manipulated without appearing in the user interface.

  For more information about creating a result set, see “Accessing data from a database using a batch process” earlier in this chapter.

Before you choose which way to find data, consider how you want to use the found records:

- If you create a found set, you can see the record data immediately in fields in the application views.
- If you create a result set, you must do one of the following with the data:
  - Modify the data through LotusScript.
  - Display data from the result set in text blocks or other display elements.
  - Create a new .APR file based on the result set.

These techniques for working with a result set are described in “Modifying records” later in this chapter.

Finding records using a Find object involves the following operations:

- Determining the find condition or conditions. To use input from the user for the find, display a form as a dialog box or use the LotusScript InputBox function.
- Creating a Find object, specifying the field to search and the value to match.
- Adding other conditions to the Find object, using the And or Or methods.

The following example prompts the user for a last name and a state name, and it creates and executes a find using the input. The script is part of the sample application stored in DW08_S2.APR in the sample files directory. The script is attached to the Click event for a button on a form.

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

' Click event for the btnLast_St object

' Apply the Find and Sort objects to the DocWindow using the
' FindSort method.
' Prompt the user to enter an employee's last name and state.
' Start search after the user enters the information.
   ' Create a DocWindow object.
   Dim MyDocWin As DocWindow

   ' Retrieve the active DocWindow.
   Set MyDocWin = CurrentApplication.ActiveDocWindow

   ' Create a Table object.
   Dim MyTable As String

   ' Retrieve the name of the first table for the document.
   MyTable = CurrentDocument.Tables(0).TableName

' Prompt the user to enter find conditions for Last name
' and State.
   Dim MyLast  As String   ' User input for Last name
   Dim MyState As String   ' User input for State

   ' Prompt user to enter employee's last name.
   MyLast = InputBox$("Enter employee's last name", , _
      ,300,300)

   ' Prompt user to enter postal code for the state.
   MyState = InputBox$("Enter state abbreviation " & _
      "(For example, CA for California)", , ,300,300)
```

```
' Check that the user enters information or does not
' press Cancel.
If MyLast <> "" And MyState <> "" Then

' Find the records that match the user's input and
' sort them in ascending order by last name and first name.

  ' Create a new instance of Find object to search by last
  ' name entered by the user.
  Dim MyFind As New Find (MyTable & ".Last", MyLast)

  ' Also find by state.
  Call MyFind.AND (MyTable & ".ST", MyState)

  ' Create new instance of Sort object, sorted by last name.
  ' The constant LTSSORTASCENDING indicates the sort
  ' direction.
  Dim MySort As New Sort (MyTable & ".Last", _
     LTSSORTASCENDING)

  ' Also sort by first name in ascending order.
  Call MySort.ADD (MyTable & ".First", LTSSORTASCENDING)

  ' Start Find/Sort.

  ' Add error handling here to check for finds that return
  ' no records.
  ' Run the find.
  Call MyDocWin.FindSort (MyFind,MySort)

  ' Show the find results in a worksheet view.
  Set CurrentWindow.ActiveView = _
     CurrentDocument.Worksheet~ 1
Else

 Exit Sub   ' Exit if user pressed Cancel or did not
            ' enter values.

End Sub   ' Click event for the btnLast_St object
```

Modifying records

There are two ways to modify records in a database using LotusScript:

- Create a result set and modify data in fields without exposing the data to the user. This method is especially useful if you are modifying a large number of records or updating records with information that isn't unique to each record or doesn't require user input.
- Change the text property of a field box, drop-down box, radio button, or other display object. This method handles each field in each record individually. It is especially suited to entering specific user input into a database.

Modifying records using a result set

This technique requires you to create a result set using Connection and Query objects as described in "Accessing data from a database using a batch process" earlier in this chapter. After you create the result set, loop through the records in the result set and make the changes.

The following example illustrates this process. The text of this script is stored in DW08_S3.LSS in the sample files directory.

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

' Click event for any button object
' * RUN-TIME DEPENDENCIES
' * Files: This script requires an ODBC database named Sample
' * in the same directory as the .APR file. The database
' * contains a table named USERID.CUSTOMER. The table
' * contains the fields State and SaleRep.
   Dim fName As String
   Dim Con As New Connection
   Dim Qry As New Query
   Dim Rs As New ResultSet
   Dim ChkSetV As Integer
   Dim MyVal As Variant
   Dim i As Integer   ' Index to table rows
   Dim TextToMatch As String

' Determine the find condition.

' Note: Here you can use some value in the current .APR to
' evaluate changes in the other file.
' For example, use "If MyVal = Val(Source.Field1.Text)  Then"

TextToMatch = "CA"
```

```
' Open a connection to the table.
If (Con.ConnectTo ("ODBC Data Sources","userid", _
   "password", "!Sample")<>False) Then

   ' Use this connection for the query.
   Set Qry.Connection = con

   ' Specify the table for the query.
   Qry.TableName = "USERID.CUSTOMER"

   ' Use this query to create the result set.
   Set Rs.Query = Qry

   ' Create the result set.

   ' If the query was successful, then...
   If ((Rs.Execute)<>False) Then

      ' Find the number of columns (fields) in the table.
      N = Rs.NumColumns

      ' Print the number of columns
      ' to the IDE Output panel.
      Print "Number of columns = ", N

      ' Print the names of each field in the table.
      For i= 1 To N

         fName = Rs.Fieldname (i)
         Print fName   ' Output appears in the IDE.
      Next

   Else   ' If the result set was not successful,
          ' warn the user.

      MessageBox "The query did not succeed.", _
        MB_OK + MB_ICONINFORMATION + MB_APPLMODAL, _
        "Unsuccessful Query"

   End If   ' If the result set was successful.
Else   ' If the connection was not successful,
       ' warn the user.

   MessageBox "Connection failed", MB_OK + _
      MB_ICONINFORMATION + MB_APPLMODAL, "Connection"

End If   ' If the connection was successful.
```

```
   ' Read the value in a field and check if it
   ' matches the value.

   ' Continue while the variable i is less than or equal to
   ' the number of rows in the table.
   For i = 1 To Rs.NumRows
      Print i

      ' Get the value of the field State.
      MyVal = Rs.GetValue("State")
      Print MyVal

      ' If the value in the field matches the find condition,
      ' then set the value of another field.
      If MyVal = TextToMatch Then

         ' Set the field "SalesRep" to the value SF.
         Rs.SetValue "SalesRep", "SF"

      End If      ' The field value matches.

      ' Update the current row and move to the next one.
      ' UpdateRow is required to commit the changes for
      ' each record.
      Rs.UpdateRow
      Rs.NextRow
   Next      ' Until there are no more rows in the table

   ' Disconnect here to avoid trouble reconnecting later.
   Con.Disconnect
End Sub
```

Modifying records using a display element

This technique assumes that the data you want to modify appears in a view. The script loops through each record in the current found set to make the modifications. This script is in the sample application stored in DW08_S4.APR in the sample files directory.

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

' Click event for the btnEnterComment Button object
' Prompt the user for input.
' Append today's date to the user's comment.
' Append the comment to each record in the found set.
' * RUN-TIME DEPENDENCIES
' * Files: This script requires a field named Note in the
' * main table associated with the .APR file.
```

```
   ' Create a DocWindow object.
   Dim MyDocWin As DocWindow

   ' Retrieve the active DocWindow.
   Set MyDocWin = CurrentApplication.ActiveDocWindow

   Dim UserInput As String        ' User input
   Dim NoteEntry As String        ' The input with
                                  ' today's date.
   Dim PreviousEntries As String  ' Existing contents of Note
   Dim WholeNote As String        ' All the data from Note
   Dim i As Integer               ' Index of found set

   ' Get input from user.
   UserInput = InputBox$("Enter your comments", , ,300,300)

   ' Check that the user entered a comment.
   If UserInput <> "" Then

      ' Append today's date to the user input.
      NoteEntry = Date$ & ": " & UserInput

      ' Loop through each record in the found set and
      ' update the Note field.
      MyDocWin.FirstRecord      ' Go to first record.

      ' Loop through the number of records in the found set.
      For i = 1 To MyDocWin.NumRecordsFound

         ' Store the existing contents of the Note field.
         PreviousEntries = Source.Note.Text

         ' Append the new entry to the existing ones.
         WholeNote = PreviousEntries & " " & NoteEntry & "."

         ' Insert the new Note in the field.
         Source.Note.Text = WholeNote

         ' Go to the next record in the found set.
         MyDocWin.NextRecord
      Next

   End If ' If the user entered a comment.
End Sub   ' Click event for btnEnterComment
```

Displaying data from a result set in a view

When you create a result set, there are two ways to show the retrieved data in an Approach view:

- Use the result set as a table associated with a new Approach document. Use this technique if you are going to use this subset of information for tasks that require user input, such as finding records with user input or building reports.

  For more information, see “Creating a document to display a result set” later in this chapter.

- Display data from the result set in text blocks or other display elements. Use this technique if you are using the information from the result set in addition to data already available in a view. For example, add text blocks containing result set information to a report that already contains information from another database.

To display information from the result set in a view, create display elements and place them in a view. As described in “The Approach object model” earlier in this chapter, you can modify display elements using these properties:

- Background
- Border
- Color
- Height, Width
- Top, Left
- Name

The script that accomplishes this task involves the following operations:

- Creating the text block (or other display element) to hold the information
- Filling the text block with the correct value from the result set
- Setting the display properties for the text block so it matches the view
- Positioning the text block in the view
- Naming the text block

The following script illustrates this process. It is part of the Approach Meeting Room Scheduler SmartMaster application. To open this SmartMaster, choose File - New Database and select the Meeting Room Scheduler.

The example sub, DisplayBlock, displays the owner of a room reservation in the correct time slot in a view that displays the reservation information for a particular day. The sub is called from another sub that creates a result set for the specified day and passes the reservation information to DisplayBlock.

```
Sub DisplayBlock(Txt As String, Start As Double, _
   Finish As Double, RoomName As String)

' Display the reservation owner in the correct time slot
' in the current view body.
' DisplayBlock is called from readBlock.
   '  Txt            reservation owner
   '  Start          reservation start time
   '  Finish         reservation end time
   '  RoomName       reserved room name or number

' * RUN-TIME DEPENDENCIES
' * Constants: Uses constants defined by LotusScript
' * in LSCONST.LSS.
' * Globals: Uses the global array Rooms() filled by the
' * readBlock sub.

   ' Declare variables.

   Dim Tt As TextBox    ' New text block to hold the
                        ' reservation owner's name
                        ' in the view.
   Dim i As Integer     ' Index of array with the room names

   ' Index of the room that matches the RoomName passed in.
   ' It is used to determine the vertical placement of the
   ' reservation in the view.
   Dim MatchedRoom As Integer

   ' Offset and multiplier for the vertical placement of
   ' the reservation.
   Dim VerticalPlacement As Integer

   ' Search through the global array Rooms to find the room
   ' passed in from the schedule database using the global
   ' sub readBlock, also part of this .APR file.
   ' Set MatchedRoom to the index of the room passed in.
   For i = 0 To UBound(Rooms)

      If Rooms(i) = RoomName Then
         MatchedRoom = i
         i = UBound(Rooms)
      End If   ' If element matches the room passed in.
   Next
   ' Set position and display for the reservation.

   ' Header in the view takes up 1635 twips, each row in
   ' the table is 330 twips tall.
   VerticalPlacement = 1635 + (330 * MatchedRoom)

   ' Create the text block to hold the reservation.
   Set Tt = New TextBox(CurrentView.Body)

   ' Fill the text block with the reservation owner's name
   ' and spaces to center the text properly.
   Tt.Text = " " + Txt + " "

   ' Set display properties so text block matches the form.
   Tt.Font.Size = 8

   ' Use Approach LotusScript constants for border style.
   Tt.Border.Style = $ltsBorderStyleNone
   Tt.Border.Left = True
   Tt.Border.Right = True
   Tt.Border.Top = False
   Tt.Border.Bottom = False

   ' Use Approach LotusScript constants for line width.
   Tt.Border.Width = $apr1point

   ' Use Approach LotusScript constants for color.
   Call Tt.Border.Color.SetRGB(COLOR_ULTRAMARINE)
   Call Tt.Background.Color.SetRGB (COLOR_50_GRAY)

   ' Set the position of the text block to correspond to the
   ' correct room and time.

   Tt.Height = 325
   Tt.Top = VerticalPlacement   ' Current offset from top
                                ' of the form

   ' Convert reservation time (passed in) to the horizontal
   ' location and length on the form.
   Tt.Left = (((start - 8) * 750) + 945)
   Tt.Width =  (750 * (finish - start))

   ' Add a prefix to the name of the text block so the
   ' ClearDisplay function can delete the reservation.
   Tt.Name = "Tt" + Str$(Tt.Top) + Str$(Tt.Left)

End Sub
```

Creating a document to display the result set

Approach documents (.APR files) are instances of the Document object. To create a new document to display a result set you must do the following:

- Create a connection to an existing table.
- Create a general query to extract records from that table.
- Create a result set to contain extracted records from the table on disk.
- Create a new .APR file using this result set.

The following example illustrates each of these steps. The text of this script is stored in DW08_S5.LSS in the sample files directory.

The CreateDocument sub creates a new document in Approach that displays the data from a result set. The result set is the main table for the new document.

```
Sub CreateDocument

' * RUN-TIME DEPENDENCIES
' * Files and paths: This script depends on an existing
' * dBASE IV database in the directory
' * C:\LOTUS\WORK\APPROACH\BLANK.DBF

' Declare the necessary variables.
Dim MyConnection As New Connection   ' Connection to a table
Dim MyQuery As New Query             ' Query used to extract
                                     ' data from the table
Dim MyResultSet As New ResultSet     ' Result set to contain
                                     ' extracted data in the
                                     ' new document
Dim MyDoc As Document                ' New Document object

' Specify the type of database (dBASE IV) to connect to as
' the source for the new document.
If MyConnection.ConnectTo("dBASE IV") Then

   ' Specify the name of the connection MyConnection that the
   ' query MyQuery uses to create the new document.
   Set MyQuery.Connection = MyConnection

   ' Specify the full path and table name to be used as
   ' source for the new document.
   MyQuery.TableName = "C:\LOTUS\WORK\APPROACH\BLANK.DBF"

   ' Add a statement to specify a subset of records to be
   ' returned in the result set here. "MyQuery.SQL = ..."
   ' Specify the result set in the new database to receive
   ' the new records from the query.
   Set MyResultSet.Query = MyQuery
   ' If the connection and query succeed, create the
   ' new document.
   If (MyResultSet.Execute)Then
      Set MyDoc = New Document(MyResultSet)
   End If   ' If the result set was successful.

   ' Error handling for a failed result set would go here.

   ' Disconnect from the source table.
   MyConnection.Disconnect
End If   ' If the connection was successful.

' Error handling for a failed connection would go here.
End Sub
```

Controlling how users enter data

LotusScript gives you control over the flow of your application by allowing you to set the keyboard focus, prompt for user input, determine which commands are available at a given time, and determine which actions happen by default.

The Meeting Room Scheduler SmartMaster application provides several examples of controlling the flow of an application. One sequence in particular executes the following operations:

- Switching to a form displayed as a dialog box when the user clicks a button
- Setting the focus on the form to indicate where user input is required
- Displaying the room reservations using the date entered
- Duplicating the dialog box closure so that both clicking the mouse and pressing **ENTER** complete the action

These operations are illustrated in following scripts. They are part of the Approach Meeting Room Scheduler SmartMaster application. To open this SmartMaster, choose File - New Database and select the Meeting Room Scheduler.

The first sub is attached to the SwitchTo event for the Enter Date form. This form is already set in the InfoBox to display as a dialog box.

```
Sub SwitchTo(Source As Form, View As View)

' SwitchTo event for the Enter Date form

' This script clears any text from the field box for date
' entry on the form. The Enter Date form is set to
' display as a dialog box.

   Source.Body.fbxDate.Text = ""

End Sub
```

The next sub processes input when the user clicks the OK button on the form.

```
Sub Click(Source As Button, x As Long, y As Long, _
   Flags As Long)

' Click event for the btnOK Button object on the Enter Date
' form. This script processes the date entered in the
' fbxDate field box.

   ' Display the schedule on the Schedule Display view.

   Call ProcessDate()  ' Global sub that checks that the date
                       ' is valid and displays the schedule
                       ' for that date.

End Sub      ' Click event for btnOK on the Enter Date form
```

The final script is almost the same as the previous one, except it runs when the user presses **ENTER** to close the dialog box instead of clicking OK. The sub is attached to the KeyDown event for the user entry text block on the Enter Date form. The **ENTER** key translates to a character code of 13.

```
Sub KeyDown(Source As FieldBox, CharCode As Long, _
   Repeats As Integer, Flags As Integer, _
   OverrideDefault As Integer)

' KeyDown event for the fbxDate FieldBox object on the
' Enter Date form

' When the user presses ENTER while the focus is on this
' field box, this script processes the date entered.

   ' If the KeyDown event returns an ENTER key
   ' (character code 13)

   If CharCode = 13  Then

      ' Display the schedule for that date.
      Call ProcessDate()  ' Global sub that checks that the
                          ' date is valid and displays the
                          ' schedule for that date.

   End If   ' If the ENTER key is used in fbxDate
End Sub     ' KeyDown event for fbxDate
            ' on the Enter Date form.
```

Changing the summaries in a report

This example uses the same report to display more than one type of summary information, according to user input. Changing the summaries in an existing report involves the following operations:

- Prompting the user for a field to group records by
- Grouping the panel on the input field
- Displaying the report in Print Preview mode

The following script illustrates these operations. This example is part of the application stored in DW08_S6.APR in the sample files directory.

When the user switches to the report, this script prompts for how to group the report summaries. The original report is columnar with summary groupings.

```
Sub SwitchTo(Source As Report, View As View)

' SwitchTo event for a Report object
' * RUN-TIME DEPENDENCIES
' * Files: The report that the script is attached to is a
' * summary report with a group-by field.

   Dim MyGrp As String    ' Store user input.
   Dim Flag As Integer    ' Indicates successful field
                          ' name entry.

' Go to the Browse environment.
CurrentApplication.ApplicationWindow.DoMenuCommand _
   (IDM_browse)

   ' Prompt the user for the field name by which the user
   ' wants to group records.
   Flag = 1
   While Flag = 1
      MyGrp = InputBox$("What do you want to group by?" _
         &"(Date, Product, RepID)"&Chr(10)&Chr(13)&"Date"& _
         Chr(10)&"Product", "Grouping", "Date", 300, 300)

      If MyGrp <>"Product" And MyGrp<>"RepID" And _
         MyGrp<>"Date" Then

         Flag = 1
         Beep
         MessageBox "Invalid Group Field." & Chr(13) & _
            "Try Again.",0+48+0+0, "Invalid Entry"
      Else
         Flag = 0
      End If
   Wend
   ' Change the current summary panels (leading and trailing)
   ' to reflect the new grouping.
   Source.Summary.GROUPBYDATAFIELD = MyGrp
   Source.Summary.MyLPanelFld.Datafield = MyGrp
   Source.Summary.MyLPanelFld.LabelText = MyGrp

   ' Go to Print Preview so that the user is prompted to sort
   ' on the new grouping and can see the results.

CurrentApplication.ApplicationWindow.DoMenuCommand _
   (IDM_Preview)

End Sub
```

Inserting and using OLE controls

If you have an OLE control (OCX) embedded in an Approach form, you can access the object's methods and properties through LotusScript as you would any other object. The following example opens an OCX Web browser, called Sax Webster Control, in the Internet World Wide Web Sites SmartMaster application, and passes it a URL string. The Webster OCX opens the Web page associated with that URL string, and the user is able to navigate through the Web site. If the Webster browser is not loaded, an error message alerts the user and informs the user how to install the OCX.

To use an OCX in Approach, you must first install the control. For more information, search on "Custom Controls" in the Approach Help Index.

```
Sub Click(Source As Textbox, X As Long, Y As Long, _
   Flags As Long)

' Click event for txtWebsterBrowser of the Found Set Report
' view to run the Webster browser

' * RUN-TIME DEPENDENCIES
' * Files: This script is part of the Internet World
' * Wide Web Sites SmartMaster application. It requires the
' * Sax Webster Control.

   Dim Rval As Integer   ' Return value

   ' If there is no browser installed, print a message
   ' to the user (segment below).
   On Error 11026 GoTo NoWebster

   ' Set the global StrURL to the current listing.
   StrURL = "http://" & source.URL.text
```

```
 ' If the First Viewed date is blank, add today's date.
   If (Source.fldFirstDate.Text = "") Then

      ' Enter today's date in the FirstDate field box.
      Source.fldFirstDate.ReadOnly = False
      Source.fldFirstDate.Text = Str(Today())
      Source.fldFirstDate.ReadOnly = True   ' Set it back.
   End If

   ' Add today's date to the Last Viewed date.
   Source.fldLastDate.ReadOnly = False
   Source.fldLastDate.Text = Str(Today())
   Source.fldLastDate.ReadOnly = True

   ' Switch to the Webster browser view.
   Set CurrentApplication.ActiveView = _
      CurrentDocument.Webster~ Browser

   ' Load the URL into the Webster OCX.
   Rval = CurrentView.Body.oleWebster.LoadPage(StrURL, 0)

   ' Leave this sub.
   GoTo EndLoadPage

NoWebster:

   ' Warn the user that the Webster connection isn't working.
   Print Err
   MessageBox "You don't have the Webster browser " & _
      "installed. Go to Main Menu-Setup to install the " & _
      "Webster browser."

   Resume EndLoadPage
EndLoadPage:
End Sub
```

Chapter 9
Using LotusScript in Freelance Graphics

Writing scripts in Freelance Graphics

You can use LotusScript to automate tasks the user does on a regular basis in Freelance Graphics. For example, you can use LotusScript and Freelance Graphics objects to create applications that make it easy to edit a frequently used presentation (by attaching scripts to a SmartMaster). You can also do the following:

- Attach a script to a page, a “Click here...” block, or an icon
- Write scripts that operate on documents, pages, and objects
- Use scripts to make presentations using information gathered from other Lotus products, such as 1-2-3

There are examples of scripts at the end of this chapter that you can use as they are, or as models for your own scripts.

Information for upgraders

This section describes the difference between scripting in Freelance Graphics 96 and Freelance Graphics 97.

Compatibility
Scripts that were created in Freelance Graphics 96 will run in Freelance Graphics 97. However, scripts written in Freelance Graphics 97 and saved in .PRZ (presentation) files or .SMC (SmartMaster with content) files will not run and cannot be edited in Freelance Graphics 96. If you are in Freelance Graphics 96 and you run an .LSS file created in Freelance Graphics 97 by choosing Edit - Script - Run, the script will work only if it does not use any of the new scripting features.

You can open a Freelance Graphics 97 file in Freelance Graphics 96. If you save a file containing scripts created in Freelance Graphics 97 file in Freelance Graphics 96, then all the scripts it contained will be lost.

If you open a Freelance Graphics 96 presentation in Freelance Graphics 97, open the Integrated Development Environment (IDE), and resave the scripts attached to that presentation, the 96 script format will be converted to the 97 format, and these scripts will no longer be seen by Freelance Graphics 96.

A Freelance Graphics 96 .LSO file will run in Freelance Graphics 97. However, a Freelance Graphics 97 .LSO file may fail in Freelance Graphics 96.

Attaching scripts to objects in .PRZ files

In Freelance Graphics 97 you can attach scripts to objects in .PRZ files as well as .SMC files. In Freelance Graphics 96 you could only attach scripts to objects in an .SMC file. Note that .SMC files are generally the most useful files in which to attach scripts to objects.

Events

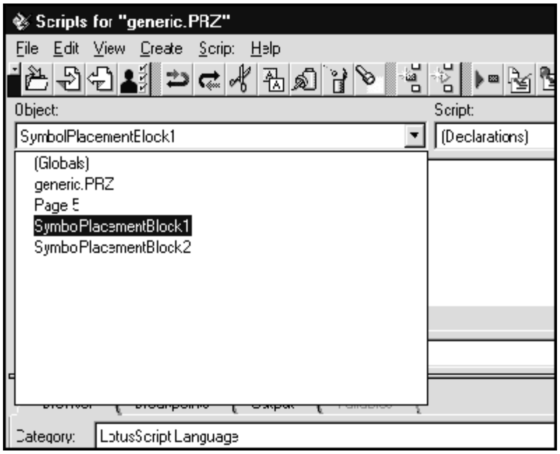
Freelance Graphics now has events for the following classes:

Class	*Events*
Document class	Activated, Created, Opened, PageCreated, PreClose, Save, SaveAs, Saved, SavedAs, SMCStarted
Page class	Activated, Created
PlacementBlock class	Clicked

For more information about Freelance Graphics events, search on “Events (LotusScript)” in the Freelance Graphics Help Index.

Objects listed in the IDE

For each page of a Freelance Graphics 97 presentation, you can now see the list of objects that have events by opening the Object drop-down box of the IDE.



Object: SymbolPlacementBlock1

Script: (Declarations)

(Globals)
generic.PRZ
Page 5
SymbolPlacementBlock1
SymbolPlacementBlock2

Category: LotusScript Language

You can use the name listed in the Object drop-down box in a script as a reference to the object it represents, but you must put square brackets around it. For example:

```
[SymbolPlacementBlock1].Insert(MyCircle1)
```

Using the name of an object as listed in the IDE is a simple way of manipulating the object in a script.

For information about how to work with those objects not listed in the IDE, see “Using names to manipulate objects,” later in this chapter.

Default object names

In Freelance Graphics 97, all instances of the DrawObject class have default names (not just instances of the page and the document classes as in Freelance Graphics 96). For more information, see “Using names to manipulate objects,” later in this chapter.

Indexing collections

In Freelance Graphics, the index value of the first element in a collection or table is 1. In other Lotus products (Word Pro and Approach, for example) the index of the first element is 0. For more information about Freelance Graphics collection classes, see “Freelance Graphics collection classes” later in this chapter.

Note You can use the LotusScript Option Base statement to change the index value of the first element in a collection. For more information on the Option Base statement, search on “LotusScript” in the Freelance Graphics Help Index, then click “LotusScript Index.”

Dialog Editor

Freelance Graphics 97 has a Dialog Editor that you use to create and save custom dialog boxes. The Dialog Editor offers more flexibility than the RunDialog method; however, the RunDialog method is still available. For information about the Dialog Editor, see Chapter 3.

Transcript window

The Transcript window was used in Freelance Graphics 96 to show error messages and output. However, in Freelance Graphics 97, error messages generated by event scripts and other scripts (run from the IDE) appear in the Errors drop-down box in the IDE.

In Freelance Graphics 97, the Transcript window is now referred to as the Output window. The Output window automatically opens when a script error happens outside of the IDE, for example, when you run a script by choosing Edit - Script - Run or when you run a script from an icon. In addition, when you run presentations containing scripts that were created in Freelance Graphics 96, the only way you can see error messages is by opening the Output window. To do so, choose Edit - Script - Show Output Window.

Printing from the IDE

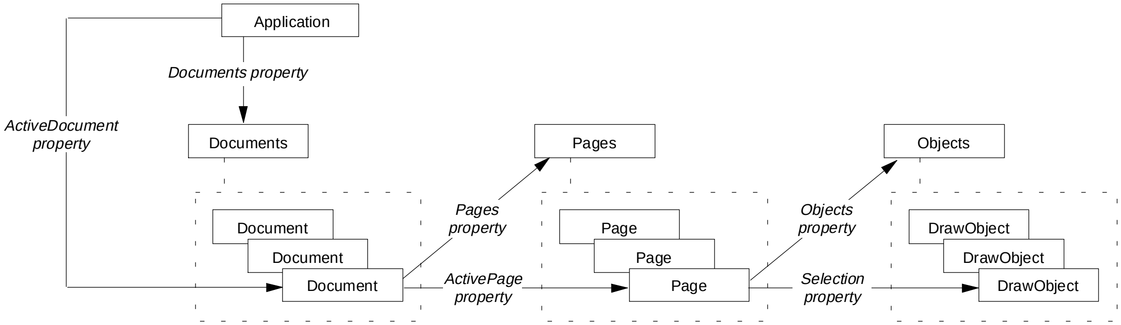
You can now print a script from the IDE in Freelance Graphics 97. From the IDE main menu, choose File - Print Script.

The Freelance Graphics object model

Before you begin writing scripts in Freelance Graphics, you should take some time to understand the Freelance Graphics object model, which describes the main Freelance Graphics objects and their organization.

Freelance Graphics containment hierarchy

The diagram below shows the main containment hierarchy in Freelance Graphics.



The containment hierarchy for Freelance Graphics begins at the top with the Application class and works its way through the familiar structure of Freelance Graphics: from .PRZ files to pages and elements on a page, such as rectangles.

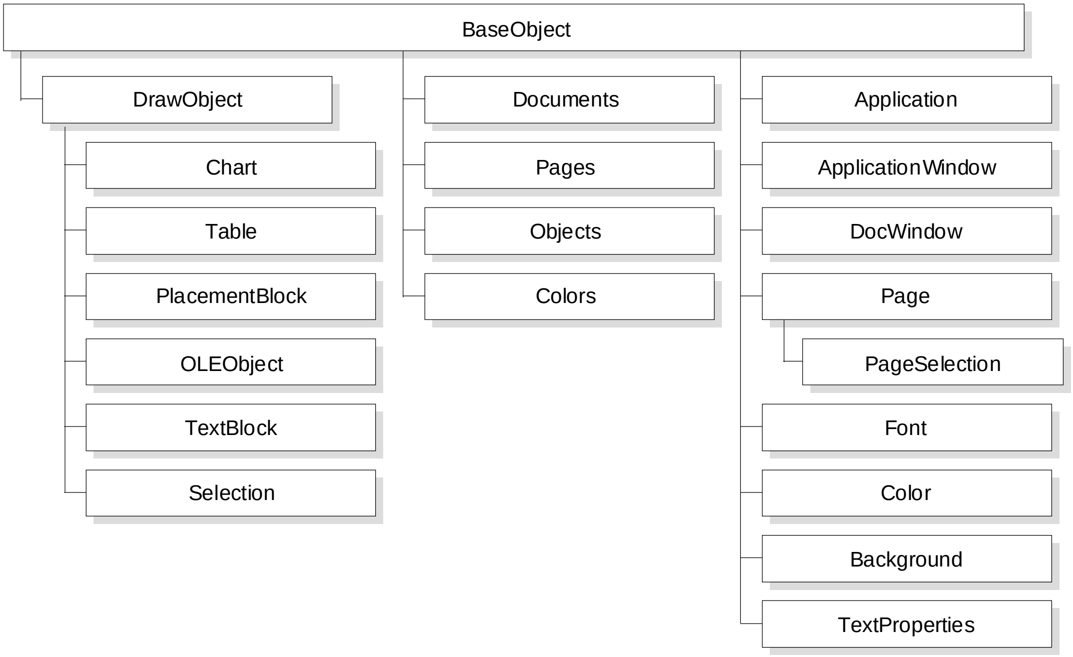
The Application class contains the Documents class (a collection class representing all of the documents currently open in the application), which is accessed through the Documents property. The Application class also contains a Document class (representing individual presentations, or .PRZ files) which is accessed through the ActiveDocument property.

The Document class, in turn, contains the Pages class (a collection class representing all of the pages in a document), which is accessed through the Pages property. The Document class also contains the Page class (representing individual pages), which is accessed through the ActivePage property.

The Page class contains the Objects class (a collection class representing all of the elements on the page), which is accessed through the Objects property. The Page class also contains the DrawObject class (representing elements on the page, such as rectangles, "Click here..." blocks, charts, and OLE objects), which is accessed through the Selection property.

Freelance Graphics inheritance relationships

The diagram below illustrates the most important inheritance relationships in Freelance Graphics.



The inheritance diagram shows most of the classes in Freelance Graphics. BaseObject is an abstract class. Most Freelance Graphics classes are derived from it. Since the BaseObject class is an abstract class, you never create instances, or objects, of that class.

The diagram also shows the classes, such as Selection and PlacementBlock, that are derived from the DrawObject class. The derived classes directly inherit the members from the DrawObject class.

Freelance Graphics collection classes

The collection classes in Freelance Graphics (Documents, Pages, Objects, and Colors) inherit from the BaseObject class. A collection class is made up of a collection of objects of that particular class.

The following table shows the types of indexes you use to access elements of Freelance Graphics collections.

Class	*Indexed by*
Documents class	Number
Pages class	Number and name
Objects class	Number
Colors class	Name

You can get the index of any object by using the GetIndex method. The index numbers correspond to the order in which the objects are created. Note that the current document is always 1. For general information about collection classes, see “Collection classes” in Chapter 2.

Note In Freelance Graphics script syntax, the index of the first item in a collection or table is 1.

Freelance Graphics predefined global product variables

Predefined global product variables let you operate on Freelance Graphics objects. The following table shows the predefined global variables you can use to specify Freelance Graphics objects:

Variable	*Description*
CurrentApplication	Represents the current session of Freelance Graphics and uses the properties and methods of the Application class.
CurrentApplicationWindow	Represents the application window of the current session of Freelance Graphics and uses the properties and methods of the ApplicationWindow class.
CurrentDocument	Represents the current Freelance Graphics document and uses the properties and methods of the Document class.
CurrentDocWindow	Represents the current Freelance Graphics document window and uses the properties and methods of the DocWindow class.
CurrentPage	Represents the current page and uses the properties and methods of the Page class.
Selection	Represents the currently selected element(s) on a page and uses the properties and methods of the DrawObject class.

For examples of how to use global variables see “Using predefined global product variables,” later in this chapter.

Using the IDE in Freelance Graphics

Open the IDE by choosing Edit - Script - Show Script Editor from the Freelance Graphics main menu. For more information about the IDE, see Chapter 3.

Using the Dialog Editor in Freelance Graphics

Open the Dialog Editor by choosing Edit - Script - Show Dialog Editor from the Freelance Graphics main menu. Freelance Graphics saves the dialog boxes you create with the Dialog Editor along with the .SMC or .PRZ file in which you created them.

For more information about using the Dialog Editor, see Chapter 3.

Customizing the Freelance Graphics user interface

You can attach scripts to pages and placement blocks ("Click here..." blocks) in an .SMC file—the standard method and the one that has greatest applicability. You can also attach scripts to objects in a SmartMaster look file (.MAS file) or a .PRZ file. In addition, you can attach scripts to SmartIcons, run unattached scripts from Freelance Graphics, and run scripts from the command line.

The advantage of attaching scripts to pages and "Click here..." blocks in an .SMC file is that .SMC files are the templates for .PRZ files. Scripts attached to .SMC files can be used each time you create a presentation using the .SMC file. However, scripts attached to a page or a "Click here..." block in a .PRZ file can only be used in that .PRZ file.

Creating "Click here..." blocks

A "Click here..." block, also known in scripting as a placement block, can be either a TextPlacementBlock, a Button, a SymbolPlacementBlock (for clip art), a ChartPlacementBlock, an OrgChartPlacementBlock, a TablePlacementBlock, or a DiagramPlacementBlock. Once created, all of these placement blocks can have scripts attached to them. You can create placement blocks only while doing one of the following tasks:

- Editing SmartMaster content files (.SMC files)
- Editing SmartMaster look files (.MAS files)
- Editing a page layout or backdrop in a presentation file (.PRZ file)

Note From the scripting point of view, the preferred method is to create placement blocks and attach scripts to them in .SMC files. Doing so provides you with the greatest flexibility.

When a placement block is no longer a placement block

A “Click here...” block is defined as a placement block as long as it has not yet been filled in by the user. When the user fills in a placement block (with text for example), from the point of view of LotusScript, the placement block is no longer a placement block (or a “Click here..” block). It is a simple text block, and a Clicked event associated with the original placement block will not trigger the execution of the script. If the user filled in the placement block with a table or an organization chart, then the placement block becomes a simple table or organization chart, and so on for the other types of placement blocks.

You cannot attach a script to a simple text block (or to a chart, a table, clip art, and so on). Once a placement block becomes a text block, the text block is recognized only as a text block. If you attempt to use a script to look for a placement block and the placement block has been filled in, the script will not find the placement block that you are asking it to look for because to LotusScript, the placement block no longer exists. You have to use the IsText property (or IsChart, or IsTable, or other related properties) not the IsPlacementBlock property, when you look for a placement block that has been filled in with text (or a chart, or a table, and so on). If you bind an object variable to a placement block, the results will be unpredictable if the placement block is filled with text.

Note When the user removes text that was entered in a placement block, then LotusScript will recognize it as a placement block again.

Attaching scripts to pages and “Click here...” blocks in an .SMC file

To attach a script to a page or a “Click here...” block in an .SMC file, choose File - Open from the Freelance Graphics main menu, then under Files of Type, select “Lotus Freelance SmartMaster Content (.SMC).” Then go to the page or create or edit the “Click here...” block to which you want to attach a script.

Scripts are associated with events. For example, when you want to attach a script to a “Click here...” block in the Script Editor, select the PlacementBlock object (such as a TextPlacementBlock, a Button, or a SymbolPlacementblock), then select the Clicked event and type a script. To attach a script to a page, follow the same procedure, except select the Page object and then select one of the events (Created event or Activated event) associated with the Page class. Scripts are saved with the .SMC file and with any .PRZ file that is based on the .SMC file.

For “Click here...” blocks, the script runs when the user clicks the “Click here...” block in a .PRZ file that uses the SmartMaster content page containing the “Click here...” block.

When you attach a script to a page using the Created event, the script runs the moment the user chooses the content page with the script attached to it.

For more information about .SMC files, search on “Designing, content topics” and “Customizing, content topics” in the Freelance Graphics Help Index. For more information about attaching scripts, search on “Attaching, scripts” in the Freelance Graphics Help Index.

Note It is possible to attach a script to an object in an .MAS file. Use the same process as you would for an .SMC file, but open a .MAS file instead.

Attaching scripts to pages or “Click here...” blocks in a .PRZ file

You can also attach scripts to objects in .PRZ files. Presentation files are the standard files that you create presentations in when you open Freelance Graphics. When you attach a script to a “Click here...” block or to a page, it is saved with the .PRZ file.

Note You can attach scripts to a “Click here...” block (such as, a TextPlacementBlock, a SymbolPlacement Block, or a Button) in a presentation file, but you cannot create a “Click here...” block in a presentation file. You must be in either an .SMC file or an .MAS file, or be editing a page layout or backdrop. For more information about editing a page layout or backdrop, search on “Editing, page layouts” or “Customizing, the backdrop” in the Freelance Graphics Help Index.

Running an .LSS file

You can use any text editor to write a script that operates on a document, page, or object. When you write a script, save it as a plain text, LotusScript (.LSS) file. You run an .LSS file in Freelance Graphics by choosing Edit - Script - Run and typing the file name or by attaching the .LSS file to an icon.

When you run an .LSS file in Freelance Graphics by attaching it to an icon or by choosing Edit - Script - Run, Freelance Graphics executes module-level code; that is, code that is not placed between the Sub and End Sub keywords in a Sub statement. For example:

```
Call SpecialPrg1
```

or

```
Print "Foo"
```

However, if Freelance Graphics does not find module-level code in the .LSS file, it looks for a sub named Main and runs it. Since the IDE does not support module-level code, naming a sub Main makes it possible to write a script in the IDE that will not cause errors and that runs when you run it using the Edit - Script - Run command and when you attach the script to an icon.

Note If you import a script with module-level code in the IDE or attempt to write module-level code in the IDE, you will get errors. You have to write scripts with module-level code in a text editor and save them as .LSS files.

Attaching a script to an icon

SmartIcons provide a quick, simple way to do many Freelance Graphics tasks. When you attach a script to an icon, it runs when the user clicks the icon.

To attach a script to an icon:

1. Choose File - User Setup - SmartIcons Setup.
2. Click Edit Icon.
3. Click the icon you want to edit in the “Available icons” box.
4. Click Attach Script.
5. Select “Run a script (*.LSS)” and either enter the name of an .LSS file or click Browse and select a file.
6. Click OK.

Note If the script you attach does not have module-level code, it must have a Main sub in it. For more information, see the preceding section.

For more information about attaching scripts to SmartIcons, search on “Attaching, scripts” in the Freelance Graphics Help Index.

Running a script from the command line

You can run an .LSS file from the command line by choosing Start - Run in the Windows task bar and typing the following:

```
C:\Freelancepath\F32main /r lsscript.lss filename.prz
```

where *lsscript.lss* is the name of the script you want to run and *filename* is the name of the presentation in which you want to run the script. The default directory for .LSS files is LOTUS\SMASTERS\FLG; for .PRZ files the default directory can be set by the user by choosing File - User Setup - Freelance Preferences, and then clicking “File locations”.

Using names to manipulate objects

Names offer a convenient way of manipulating Freelance Graphics objects and what they represent. In Freelance Graphics objects of Application class, Document class, Page class, and DrawObject class have default names. Objects of the Font, Background, Border, and LineStyle classes, and chart class, do not have default names.

A Document object name is the file name and is read-only. A DrawObject object has a default descriptive name, such as PlacementBlock1 or Rectangle1, which can be changed through a script. The name of a page is shown in the InfoBox or the IDE; it can be changed by editing the name in the InfoBox or by writing a script.

Note “Page 1” is the default name of the first page created in a presentation, “Page 2” is the name of the next page, and so on.

When objects are created, they are given names by default. You can change these names with scripts. Freelance Graphics stores the names given to the objects in the presentation, so that they are available in future script sessions. You can assign to a variable a value that identifies an existing page. For example:

```
Dim Pg As Page
Set Pg = CurrentDocument.Pages("Page 1")
```

To change the name of a page using a script, do the following (continuing with the above example):

```
Pg.Name = "Agenda"
```

Names provide a useful way of identifying an object so that it can be manipulated with a script. Application objects and objects with events are listed in the IDE, but page elements are not. To find out the names of all of the page elements that you can manipulate with a script, run the following script:

```
' Find the names of all elements on this page.
ForAll Obj in CurrentPage.Objects
     Print Obj.Name
End ForAll
```

Note Print output appears in the Output panel of the IDE.

For example, if you have several elements on a page, such as a text block, a rectangle, and an ellipse, and you want to manipulate the rectangle, you would use the preceding script to learn that the rectangle is named

Rectangle1. You can find any drawn object if you know its name by using the FindObject method. Once you find an element, you can manipulate it. If you want to move the rectangle, write a script such as the following:

```
Set Rect = CurrentPage.FindObject("Rectangle1")
Rect.Move 1000, 1000
```

You can even change the element name. For example:

```
Rect.Name = "Euclid"
```

Once you change the name of an element, you must refer to it by the new name. In the above example, once you renamed “Rectangle1” to “Euclid,” you would have to refer to the rectangle as “Euclid” the next time you used the FindObject method, unless, of course, you change the name again.

Freelance Graphics stores the names given to pages and elements in the presentation so that they are available in future script sessions.

You can use the Bind keyword to bind an object variable to an instance of the Document, Page, and DrawObject classes. For example:

```
Dim p As Page
Set p = Bind("Page 1")
Print p.Number
```

Using predefined global product variables

You can use predefined global product variables to write scripts that operate on the current application, application window, document window, document, or page, or the currently selected element on a page (such as a rectangle). Freelance Graphics always maintains a valid value for the global product variables.

The following script changes the pattern of the currently selected element or elements. Selection is a predefined global product variable that represents the currently selected element or elements on a page.

```
Sub Main
   Selection.Background.Pattern = $LtsFillGray2
End Sub
```

The following script gives a screen show transition delay of ten seconds to the currently selected page. CurrentPage represents the current page and uses the properties and methods of the Page class.

```
Sub Main
   CurrentPage.PageTransitionDelay = 10
End Sub
```

Using global product variables to assign values to object variables

Use global product variables as a convenient way to access all other objects. The Set statement must be used any time an assignment statement involves an object. Note that the Pages and Objects classes illustrated in the following example are collections and can be indexed like arrays. For example:

```
Dim MyRect As DrawObject
Dim MyPage As Page
' Set MyPage equal to the third page in the presentation,
' and MyRect to the second item on MyPage.
Set MyPage = CurrentDocument.Pages(3)
Set MyRect = MyPage.Objects(2)
```

Notice that in the example the global product variable CurrentDocument is used to refer to the current document and that the collection property, Pages, is used to refer to the third page of the document. Once you use the global product variable, CurrentDocument, to assign a value to the object variable MyPage, the example shows how you then use MyPage to refer to a specific element on a page.

In the example that follows, note the use of the global product variable CurrentPage in combination with the CreateRect method, to assign a value to the object variable, Rect1:

```
Dim Rect1 As DrawObject
' Create a rectangle of default size, then name it MyRect1.
Set Rect1 = CurrentPage.CreateRect
Rect1.Name = "MyRect1"
```

Later in a script you could use the name MyRect1 which you gave to the rectangle in the preceding code example, to find the rectangle so that you can manipulate it in some way. You can find an existing named page element (for example, MyRect1) on the current page by using the global product variable CurrentPage and the FindObject method (a Page Class method that CurrentPage can use).

```
Dim Obj1 As DrawObject
' Find a rectangle named MyRect1.
Set Obj1 = CurrentPage.FindObject("MyRect1")
```

To continue with this example, if you wanted to move the rectangle once you found it, you could use the following statement. The statement makes use of the object variable Obj1 that was set in the example above and the Move method (a method of the DrawObject class, since Obj1 is an instance of that class).

```
Obj1.Move 1000,1000
```

Creating elements and objects

To create a Freelance Graphics element, you must use the appropriate method. In general, you create an element by using a method of the appropriate class. You have already seen some examples of creating elements in the preceding section. In this section you will find more examples.

The methods for creating elements on a page belong to the Page class. For instance, the following example creates a rectangle:

```
Dim MyRect As DrawObject
Set MyRect = CurrentPage.CreateRect (2000, 3000, 3000, 4000)
```

In this example, you create the rectangle, MyRect, by using the global product variable CurrentPage (it uses the methods and properties of the Page class) and the CreateRect method (a Page class method) to create the rectangle. The numbers in parenthesis give the location and size and width of the rectangle on the page. For more information about how to use the CreateRect method, search on “Methods (LotusScript)” in the Freelance Graphics Help Index.

The method for creating a page belongs to the Document class; therefore you can use it with the global product variable CurrentDocument. In the following example, the CreatePage method takes two parameters, the title of the page and the SmartLook (or template) on which the page will be based.

```
Dim MyPage As Page
Set MyPage = CurrentDocument.CreatePage("Title of Page", 2)
```

The NewDocument method for creating a document belongs to the Application class. The following script creates a new document.

```
CurrentApplication.NewDocument
```

Note You must save a document to name it.

Also, the OpenDocument method opens an existing document. For example, to open the existing presentation, PROPOSAL.PRZ, do the following:

```
Set MyDoc = CurrentApplication.OpenDocument("proposal.prz")
```

You use the NewDocument method to create a presentation and the OpenDocument method to open a presentation.

Top tasks

All of the examples in this section should be used in .SMC files. Some may work in .PRZ files, but for general applicability, attach your scripts to objects in .SMC files. The code for each example is in a file in the sample files directory. The file name for each code example is given before each example.

Putting clip art on the current page

The text of this script is stored in DW09_S1.LSS. To save time, import the file into the IDE by choosing File - Import Script.

This script defines a sub named LaunchTextBox that puts a piece of clip art on the current page. Note that you can attach this script to the Click event of a button.

```
Sub LaunchTextBox

' This sub uses the CreateSymbol method (Page class) to
' create clip art. In this example, one of the
' parameters of CreateSymbol uses the TemplateDir property
' of Preferences (Preferences is itself a property of the
' Application class) to specify the path for the clip art
' file. The file name is concatenated to the path.
' The second parameter indicates which of the text box
' symbols to add.

   CurrentPage.CreateSymbol _
      CurrentApplication.Preferences.TemplateDir + _
      "textbox.sym", 1

   ' Once the clip art is created, it is a selected element
   ' on the page. The next line of code uses the
   ' ClearSelection method to deselect all
   ' elements on the page. ClearSelection is a method of
   ' both the Selection and PageSelection classes.

   Selection.ClearSelection

End Sub
```

Using an event

The text of this script is stored in DW09_S2.LSS. To save time, import the file into the IDE by choosing File - Import Script.

This script uses an event to run a script that prompts the user for a title for a page whenever the user creates a page from the .SMC page that this script is attached to. The script is executed each time a page is created. This script asks the user for a page title and puts it in the title placement block.

Note You must attach this script to the Created event of a page in an .SMC file for it to be of use.

```
Sub Created(Source as Page)

   Dim Title As String
   Dim MyText As DrawObject

   ' Use the LotusScript InputBox function to create a
   ' dialog box with a prompt for user input.

   Title = InputBox$ ("Please type a page title ", _
               "Page Title")

   ' Create a text block that will hold the text
   ' that the user types.

   Set MyText = CurrentPage.CreateText

   ' Put the text that the user types into the text block.

   MyText.Text = Title

   ' Move the text block into the "Click here..." block by
   ' using the PutIntoPlacementBlock method—this is
   ' a method of the DrawObject class.

   MyText.PutIntoPlacementBlock(1)

End Sub
```

Printing the current page

The text of this script is stored in DW09_S3.LSS. To save time, import the file into the IDE by choosing File - Import Script.

This script opens a dialog box that prompts the user to indicate whether the current page should be printed. It also demonstrates how to use the file LSCONST.LSS, which contains a number of predefined LotusScript constants, such as MB_OKCANCEL and IDOK.

To make use of LSCONST.LSS, use it as an argument to a Use statement in an (Options) script.

```
Use "lsconst"
Sub PrintCurrentPage()

   Dim PrintRetVal As Integer

   ' Use the LotusScript MessageBox function to create a
   ' dialog box with a prompt for user feedback.
   PrintRetVal = Messagebox("Would you like to print " + _
      "this page?", MB_OKCANCEL, "Print")

   ' Assess the user response to the message box; if the user
   ' clicks OK, then print the current page.
   If(PrintRetVal = IDOK) Then
         CurrentDocument.Print CurrentPage.Number, _
             CurrentPage.Number, 1
   End If

   Selection.ClearSelection

End Sub
```

Launching a clip art or diagram browser with a specified file

The text of the scripts described here is stored in DW09_S4.LSS. To save time, import the file into the IDE by choosing File - Import Script.

There are two examples in this section. One shows how to write a script that opens the clip art browser with a specified file, the other script shows how to open the diagram browser with a specified file.

For this script to work, the .SMC file must have a symbol placement block or a diagram placement block on the page. You can attach the script to the page, the Click event of a button, or the Click event of the placement block itself.

Launching the clip art browser

This script consists of two subs. The first is LaunchSymBrowser, the primary sub that actually opens the clip art browser. The second sub is the sub that calls LaunchSymBrowser with a parameter indicating which clip art file the browser should open. Once the browser is open, the user selects which item to place on the page.

The sub LaunchSymBrowser looks for an existing placement block (a SymbolPlacementBlock in this case) on the current page and then opens the clip art browser so that the user can select clip art for that placement block.

```
Private Sub LaunchSymBrowser(SymbolFile As String)

   Dim SymPB As DrawObject

   ' The following line locates the placement block named
   ' SymbolPlacementBlock1 on the current page and
   ' then assigns it to the object variable, SymPB.
   ' To do this, you have to know the name of
   ' the placement block.

   Set SymPB = _
      CurrentPage.FindObject("SymbolPlacementBlock1")

   ' The next line uses the BrowseSymbols method of the
   ' PlacementBlock class to open the symbol browser
   ' with the name of the clip art file that
   ' has been passed to this sub. Note the use of
   ' the TemplateDir property to indicate the path to
   ' the file.

   SymPB.PlacementBlock.BrowseSymbols _
      CurrentApplication.Preferences.TemplateDir + SymbolFile

   ' Note: you could write a variation on this sub that first
   ' creates the placement block, positions it on the page,
   ' and then opens the symbol browser.

End Sub

' The following sub, PeopleSymbol, calls the
' LanuchSymBrowser sub with the name of the clip art file
' that it will use.

Sub PeopleSymbol()

   LaunchSymBrowser("people.sym")

End Sub
```

Launching the diagram browser

The following sub LaunchDgmBrowser does the same thing as described in the previous example. However, in this case, the sub opens the diagram browser.

The sub, LaunchDgmBrowser, looks for an existing diagram placement block on the current page, then opens the diagram browser so that the user's selection will go into that placement block.

```
Private Sub LaunchDgmBrowser(DiagramFile As String)

   Dim DgmPB As DrawObject
   Set DgmPB = _
      CurrentPage.FindObject("DiagramPlacementBlock1")

   ' The next line uses the BrowseDiagrams method of the
   ' PlacementBlock class to open the diagram browser.

   DgmPB.PlacementBlock.BrowseDiagrams _
      CurrentApplication.Preferences.TemplateDir + _
         DiagramFile

End Sub

' The following sub, GanttDiagram, calls the
' LaunchDgmBrowser sub with the name of the file that
' contains the diagram that is used in this example.

Sub GanttDiagram()

   LaunchDgmBrowser("gantt.dgm")

End Sub
```

Filling a bulleted list with text

The text of this script is stored in DW09_S5.LSS. To save time, import the file into the IDE by choosing File - Import Script.

This script fills a bulleted list with predefined text. It consists of three subs: SetAgenda (the primary sub), SampleAgendaPlan, and Main. The sub Main is used so that this script can be attached to an icon as well as to an event. The code for all three subs follows one after the other.

```
' Declare globals.
' Declare a constant containing the text that will be
' used to fill the bulleted list. The text contains the
' markup sequence <= for carriage return.

Const SampleAgendaTxt = "First bullet<=" + _
   "Second bullet<=Third bullet<=Fourth " + _
   "bullet<=Fifth bullet"
```

```
' The following sub, SetAgenda, is written with the
' assumption that there is only one placement block
' on the page and that it will be turned into a
' text block.

Private Sub SetAgenda(TextToInsert As String)

   Dim AgendaTxtBlk As DrawObject
   Dim DummyTextBlk As DrawObject

   ' Using a ForAll and an If statement, search the
   ' current page for a placement block.

   ForAll Objs In CurrentPage.Objects

      If (Objs.IsPlacementBlock) Then

      ' The variable, AgendaTxtBlk, is set to be
      ' the instance of the PlacementBlock class
      ' found on the current page.

         Set AgendaTxtBlk = Objs

      End If

   End ForAll

   ' Create a temporary text block.

   Set DummyTextBlk = _
      CurrentPage.CreateText(1000,1000,1000,1000)

   ' Enter text in the temporary text block using
   ' the Text property of the TextBlock class. TextBlock is
   ' also a property of the DrawObject class. It is
   ' simpler to move a text block into a placement
   ' block than it is to move a string of text into a
   ' placement block, so this script uses this method of
   ' manipulating text.

   DummyTextBlk.Textblock.Text = "temp"

   ' Put the temporary text into the placement block using
   ' the Insert method of the PlacementBlock class.

   AgendaTxtBlk.Insert DummyTextBlk

      ForAll Object In CurrentPage.Objects
```

```
         ' Check each object to see which is the one with
         ' the text "temp" in it. When you find it,
         ' put the text you want to insert into it,
         ' then give it bullet properties.
         If (Object.TextBlock.Text = "temp") Then

            Object.TextBlock.Text = TextToInsert

            Object.TextBlock.BulletProperties.Style = _
               $ltsBulletLargeDot

            Exit ForAll

         End If

      End ForAll

End Sub

' This sub, SampleAgendaPlan, calls the sub SetAgenda
' with the text constant, SampleAgendaTxt.

Sub SampleAgendaPlan()

   SetAgenda(SampleAgendaTxt)

End Sub

' This sub, Main, calls SampleAgendaPlan. When you use the
' sub Main, the sub is generically available
' in Freelance Graphics, so that you
' can run the sub even when it is not attached to an
' event. It can, for example, be attached to an icon.

Sub Main

   SampleAgendaPlan

End Sub
```

Converting text to table entries

The text of this script is stored in DW09_S6.LSS. To save time, import the file into the IDE by choosing File - Import Script.

This script converts text into entries in a table. It consists of two subs: Main and TableInsert. Main finds the first non-empty text block on the page and converts it into a table. In this implementation, a new line creates a new row and a tab creates a new column.

Because the sub is called Main, it can be run from an icon as well as by choosing Edit - Script - Run. Alternatively, you could attach the sub to an event, such as the Click event of a placement block.

The sub Main calls another sub, TableInsert, to do some work for it. The code for TableInsert follows this sub.

```
Sub Main

   ' Declare variables.

   Dim CurrentRow As Integer
   Dim CurrentCol As Integer
   Dim MyText As String
   Dim MyCellText As String
   Dim MyTable As Table
   Dim MyTextB As TextBlock
   Dim MyBlockId As Integer

   ' Check if there is one text block that is not empty
   ' on the page. Do this by looping through all
   ' objects on the page and stopping when you find
   ' a text block that has something in it.

   ForAll MyOb In CurrentPage.Objects

      ' If you find a filled in text block,
      ' set MyTextB equal to it and exit the
      ' ForAll statement.

      If (MyOb.IsText And Len(MyOb.TextBlock.Text) > 0) Then
         Set MyTextB = MyOb
         Exit ForAll
      End If

   End ForAll

   ' If you have found nothing, then
   ' put up a message box letting the user know
   ' there is nothing valid on the page and exit the sub.
   ' Note, NOTHING is a predefined LotusScript constant; it
   ' is the initial value of an object variable.

   If (MyTextB Is NOTHING) Then
      MessageBox "There are no good text blocks on the page."
      Exit Sub
   End If

   ' Initialize the variable MyText to the text
   ' contained in the text block MyTextB.
   MyText = MyTextB.Text

   ' Create the table and fill it with the data.
   ' Initialize variables.
   CurrentRow = 1
   CurrentCol = 1
```

```
' Create the table, setting the rows and columns equal
' to one. Also, select the type of table.

Set MyTable = _
   CurrentPage.CreateTable(1,CurrentRow,CurrentCol)

' Figure out how many rows and columns you need.
' Do this by counting newlines.

' Parse the text string, MyText, using the length
' of the text string as the limit of the For loop, and
' going through each character until a carriage return or
' a tab is encountered. When one of these
' characters is encountered, call the
' sub TableInsert to put the characters found
' up to that point into a table cell. If necessary,
' the sub TableInsert also expands the table.

For i = 1 To Len(MyText)
   CurrentChar = Mid$(MyText,i,1)

   ' Is the current character a carriage return?
   ' If it is a carriage return, insert text, then
   ' go to next row.
   If (CurrentChar = Chr$(13)) Then
      ' You call the sub TableInsert with the arguments
      ' it requires. Note, Trim$ removes the leading and
      ' trailing spaces from a text string (in this case,
      ' the text contained in the string MyCellText).
      Call TableInsert(MyTable, _
         Trim$(MyCellText),CurrentRow,CurrentCol)
      CurrentCol = 1
      ' Increment row number; that is, go down to the
      ' next row.
      CurrentRow = CurrentRow + 1
      ' Reinitialize the text string variable MyCellText
      ' to the empty string.
      MyCellText = ""

   ' Is it a tab?  If so, follow same steps
   ' as above, but, after inserting text,
   ' instead of moving down, stay in the same row
   ' and move to the next column.
   ElseIf (CurrentChar = Chr$(9)) Then

      Call TableInsert(MyTable, _
         Trim$(MyCellText),CurrentRow,CurrentCol)

      ' Increment column number, that
      ' is, move to next column.
      CurrentCol = CurrentCol + 1
      MyCellText = ""
      ' If neither, then keep on assembling string.
      ' Also, make sure the character isn't a standard
      ' newline (\n, or ASCII 10).

      ElseIf (CurrentChar <> Chr$(10)) Then

       ' Assemble string by adding valid characters.
        MyCellText = MyCellText + CurrentChar

      End If

   Next i

   ' Don't forget to fill the very last cell in the table.

   If (Len(Trim$(MyCellText)) > 0) Then

      Call TableInsert(MyTable, _
         Trim$(MyCellText),CurrentRow,CurrentCol)

   End If

   ' Finally, delete the text block you found at the
   ' beginning of this script and put the table into
   ' the placement block that originally held the text block.

   MyBlockID = MyTextB.PlacementBlock.Id
   MyTextB.Remove
   MyTable.PlacementBlock.PutIntoPlacementBlock(MyBlockId)

End Sub

' This sub, TableInsert, works with the above sub, Main.
' The job of TableInsert is to insert text into a table
' cell and, if it is necessary, to expand the number of
' rows and columns in the table.

' The table that was created in Main is passed to it, as is
' the text string and the projected number of rows and
' columns in the table. When the projected number of rows
' and columns in the table is more than the actual number,
' this sub adds a row or a column to the table.

Sub TableInsert(ATable As Table, SomeText As String, _
   RowNum As Integer, ColNum As Integer)

   ' Declare variable.
   Dim MyTableCell As TextBlock

   ' Check to see if you were passed a NOTHING reference.

   If (ATable Is NOTHING) Then
      Exit Sub
   End If
   ' Expand the table if necessary—both columns and rows.

   While(ATable.RowCount < RowNum)
      ATable.InsertRow(ATable.RowCount + 1)
   Wend

   While(ATable.ColCount < ColNum)
      ATable.InsertCol(ATable.ColCount + 1)
   Wend

   ' Now, insert the text into the table cell.
   Set MyTableCell = ATable.GetCell(RowNum,ColNum)
   MyTableCell.Text = SomeText

End Sub
```

Putting information in a table into an agenda format

The text of this script is stored in DW09_S7X.LSS. To save time, import the file into the IDE by choosing File - Import Script.

The sub CreatePagesFromAgenda turns information in a table into agenda or “to do” pages, with the name of the person responsible for each task and the task as the title of a page, and the due date in the top right corner of the page. Each page has a bulleted list to be filled in by the person responsible for the task. CreatePagesFromAgenda is followed by the Main sub, that allows you to call CreatePagesFromAgenda from an icon or from the menu by choosing Edit - Script - Run. You could also attach the script to an event and avoid using the sub Main.

CreatePagesFromAgenda is designed to work with a three-column table containing the following information: The first column contains the date that the task is to be completed; the second column contains the task; and the third column contains the name of the person responsible for completing the task. This sub is set up to accommodate as many as 40 tasks. You can adjust that number to suit your needs. Also, you can have more columns containing additional information. If you do, you will have to adjust the sub as necessary.

```
' Declare global constants. These constants are text
' strings as well as numerical values. In the case of the
' numerical values, these constants are contained in
' the file LSCONST.LSS. You can use a Use
' statement to make use of LSCONST.LSS and avoid having to
' declare these constants. However, in the interests of
' completeness, the constants are explicitly
' declared in this script.

Const CreateAgendaPagesMessage  = "A page will " + _
   "be created for each agenda item you " + _
   "entered in the table. Press OK to continue."
```

```
Const CreateAgendaPagesTitle  = "Create Agenda Pages"

Const ColumnError1  = "Agenda pages can't be " + _
   "created because the columns in the table " + _
   "have been modified."

Const ColumnError2  = "The table must have " + _
   "three columns."

Const EmptyTable = "One or more cells are" + _
   "empty. If you continue, the results may " + _
   "not be what you expect. " + _
   "Click Yes to continue or No to quit."

Const NoTable = "There is no table on this page. " + _
   "The script cannot run."

Const NoTitle = "There is no title for the " + _
   "table column. This script will not run without one."

Const ErrorMsg  = "Error"

' Use this constant to identify
' which SmartMaster template you want.
Const TemplateIndex  = 2

' IDOK stands for OK button clicked.
Const IDOK = 1

' IDNO stands for no button clicked.
Const IDNO = 7

' MB_OK stands for a message box with only an OK button.
Const MB_OK = 0

' MB_OKCANCEL stands for a message box
' with OK and Cancel buttons.
Const MB_OKCANCEL = 1

' MB_YESNO stands for a message box
' with Yes and No buttons.
Const MB_YESNO = 4

' MB_ICONSTOP stands for a message box
' with a "Stop!" icon in it.
Const MB_ICONSTOP = 16

' MB_DEFBUTTON2 stands for a message box with
' the No button as the selected button when the Yes/No
' message box opens.
Const MB_DEFBUTTON2 = 256

Sub CreatePagesFromAgenda()
```

```
' Declare the variables that you need for this script.
Dim AgendaTableObj As Table
Dim Cell As TextBlock
Dim TitleTextBlk As TextBlock
Dim DateTextBlk As DrawObject
Dim TitlePB As PlacementBlock
Dim AgendaText(2 To 16, 1 To 8) As String
Dim PageTitle As String
Dim ColumnMessage As String
Dim Row As Integer
Dim Col As Integer
Dim NumRows As Integer
Dim NumCols As Integer
Dim NumChars As Integer
Dim RetVal As Integer
Dim MaxRows As Integer
Dim TableXists As Integer
Dim IsCellEmpty As Integer
Dim NoColumnTitle As Integer

' Put up message box that explains to user what
' this script does and ask if the user wants to
' let the script run to completion, also initialize
' the variable, RetVal, to the return
' value of MessageBox.
RetVal = MessageBox(CreateAgendaPagesMessage, _
   MB_OKCANCEL, CreateAgendaPagesTitle)

' Deselect all selected items on the current page.
Selection.ClearSelection

' Initialize variable.
TableXists = 0

' Check to see if user clicked OK in message box.
' If OK was clicked, then begin.
If(RetVal = IDOK) Then

   ' Check to see that there is a table on the
   ' page. If found, set flag TableXists.
   ForAll Obj In CurrentPage.Objects

      If (Obj.IsTable) Then
         ' Assign the table to the
         ' variable AgendaTableObj.
         Set AgendaTableObj = Obj
         TableXists = 1
      End If

   End ForAll
```

```
' If no table was found then, notify user and
' exit script.
If (TableXists = 0) Then
   MessageBox NoTable, MB_OK, ErrorMsg
   Exit Sub
End If

' Initialize NumCols to the number of columns
' in the table.
NumCols = AgendaTableObj.ColCount

' Check to see if user added/deleted columns,
' because addition or deletion will result in
' unpredictable results. If number of columns
' was changed, then open message box
' informing user of unpredictable results.
If(NumCols <> 3) Then
   ' Assemble message with line break spacing.
   ColumnMessage = ColumnError1 + Chr$(10) + _
      Chr$(10) + ColumnError2

   ' Open message box with the message you
   ' just assembled.
   MessageBox ColumnMessage, MB_OK, ErrorMsg

Else

   ' If number of columns is correct, begin
   ' processing. Determine number of rows in
   ' table and initialize MaxRows to it.
   NumRows = 1
   MaxRows = AgendaTableObj.RowCount
   ' Set variable Cell to the first cell, that is,
   ' the table column title cell.
   Set Cell = AgendaTableObj.GetCell(NumRows, 1)

   ' Check that column title is filled in.
   ' Note, you could look for specific text
   ' at this point rather than simply checking that
   ' the cell is not empty.
   If (StrCompare(Cell.Text, "") = 0 ) Then
      ' Open message box explaining that
      ' script will not run without a column
      ' title. Initialize NoColumnTitle to the return
      ' value of the message box, that is, when
      ' user clicks the OK button, MessageBox
      ' returns a value of 1.
      ' When user clicks the OK button, exit script.
      NoColumnTitle = MessageBox(NoTitle, MB_OK + _
         MB_ICONSTOP, ErrorMsg)
   If (NoColumnTitle = IDOK) Then
      Exit Sub
   End If

End If

' As long as the table cell is not empty and
' the variable NumRows does not exceed the
' number of actual rows in the table, begin
' processing the table cells to check that they
' all contain text.

While((Not StrCompare(Cell.Text, " ")) And _
      (NumRows < MaxRows))

   ' Increment the row count, so that you begin
   ' processing text with the second row
   ' (the first row contains column titles).
   NumRows = NumRows + 1
   Set Cell = AgendaTableObj.GetCell(NumRows, 1)

   ' Check to see if a cell is empty. If it is
   ' empty, open a message box, then exit sub.
   ' The logic here is similar to that of the
   ' previous message box code.
   If (StrCompare(Cell.Text, "") = 0) Then

      IsCellEmpty = MessageBox _
         (EmptyTable, MB_YESNO+MB_ICONSTOP + _
            MB_DEFBUTTON2, ErrorMsg)

      If (IsCellEmpty = IDNO) Then
         Exit Sub
      End If

   End If

Wend

' Cycle through the rows of the table and put text
' into a string array.
For Row = 2 To NumRows

   ' Get cell text and put it in variable Cell.
   Set Cell = AgendaTableObj.GetCell(Row, 1)

   ' Check that cell does not contain only spaces.
   ' If it does, then skip it: this will result
   ' in blank entries.
   If(Not StrCompare(Cell.Text, " ")) Then

      ' Cycle through the columns in each row
      ' and put contents in array AgendaText.
      ' Note, AgendaText has been declared
      ' as an array containing 120 elements.
      ' That means it can process a table with
      ' forty rows—for a larger table, adjust
      ' the bounds of the array or use a
      ' dynamic array.
      For Col = 1 To AgendaTableObj.ColCount
         Set Cell = AgendaTableObj.GetCell(Row, Col)
         AgendaText(Row, Col) = Cell.Text
      Next Col

   End If

Next Row

' For each row, concatenate the text from columns
' two and three. Put the concatenated text
' into the title placement block on the new page.
For Row = 2 To NumRows

   ' Initialize the variable PageTitle.
   PageTitle = ""

   ' Cycle through columns two and three
   ' of the current row.
   For Col = 2 To NumCols

      ' If at last column (column three),
      ' add text of column three to
      ' the variable PageTitle (that is, add it to
      ' the text from column two) or else
      ' take text from column two and put it in
      ' variable PageTitle.
      If(Col = NumCols) Then
         PageTitle = PageTitle + AgendaText(Row, _
            Col)
      Else
         PageTitle = PageTitle + AgendaText(Row, _
            Col) + ", "
      End If

   Next Col

   ' Create a new page for each row in the table
   ' using the bulleted list SmartMaster look
   ' and use the contents of the variable
   ' PageTitle for the page name.
   CurrentDocument.CreatePage PageTitle, 2

   ' Make a dummy text block. Set variable
   ' TitleTextBlk to it.
   Set TitleTextBlk = CurrentPage.CreateText(1000, _
      1000, 1000, 1000)
```

```
            ' Put the concatenated text from columns
            ' two and three into dummy text block.
            ' Note, it is easier to manipulate text that is
            ' in text blocks than it is to manipulate text
            ' strings. Therefore, this script uses
            ' text blocks to manipulate text.
            TitleTextBlk.Text = PageTitle

            ' Search current page to find the "Click here..."
            ' block (placement block) and assign it to the
            ' variable TitlePB.
            ForAll Obj In CurrentPage.Objects

               If (Obj.PlacementBlock.PromptText = _
                     "Click here to type page title") Then
                  Set TitlePB = Obj
               End If

            End  ForAll

            ' Insert text from dummy text block into
            ' the "Click here..." placement block.
            TitlePB.Insert TitleTextBlk

            ' Deselect the selected items on the page.

            Selection.ClearSelection

            ' Create a text block in the top right
            ' corner of the page where the due date,
            ' from column one, will be inserted. Set
            ' the variable DateTextBlk as the handle
            ' to the text block.
            Set DateTextBlk = _
               CurrentPage.CreateText(12000,10500,1000,1000)

            ' Insert text from column one into the
            ' due date text block.
            DateTextBlk.TextBlock.Text = "Due: " + _
               AgendaText(Row, 1)

            ' Deselect the text block.
            Selection.ClearSelection

         Next Row

      End If

   End If

End Sub

' The following sub, Main, makes it possible for
' you to run the CreatePagesFromAgenda sub when you
' choose Edit - Script - Run, or to run the
' script from an icon; in both cases, Freelance Graphics
' looks for a sub named Main to execute.
Sub Main
   CreatePagesFromAgenda
End Sub
```

Chapter 10
Using LotusScript in Word Pro

LotusScript and Word Pro

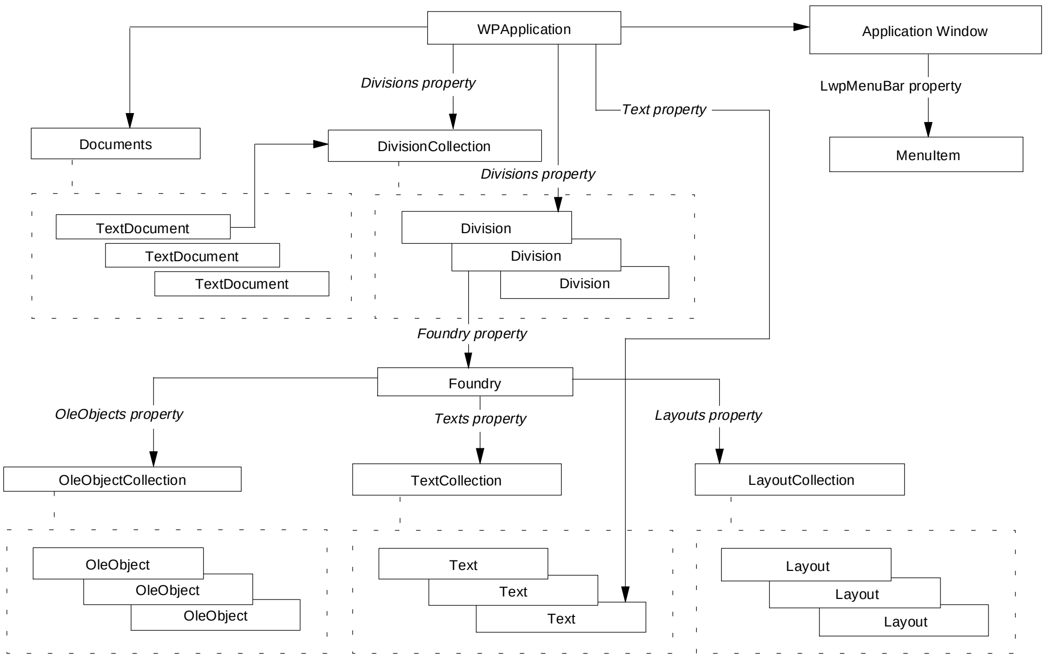
Word Pro provides a powerful set of objects, such as documents, divisions, and SmartMaster templates, that you can program using LotusScript. You can use LotusScript in Word Pro to automate frequent or repetitive tasks or to develop innovative applications built on Word Pro features. Some of the tasks you can accomplish with LotusScript in Word Pro include the following:

- Automating documents using a Word Pro SmartMaster
- Creating document views
- Displaying custom dialog boxes
- Managing find and replace options
- Managing merge operations
- Performing sort operations
- Performing special tasks when starting Word Pro
- Validating Click Here Blocks

The Word Pro object model

Word Pro has a large and comprehensive object model that provides many different ways to accomplish any given task. For general information on SmartSuite product object models, see Chapter 2.

Because the Word Pro object model is so large, the following diagram depicts only a part of the object model. It shows some of the more frequently used containment relationships you need to understand when you work with LotusScript and Word Pro.



When you write a script, you can access and control any part of Word Pro that is represented by a class in the Word Pro object model. You can create as many objects as you want from the same class; however, each object that is instantiated from a class has properties with unique values, such as the value of the Name property. For example, if you create a DocWindow object from the DocWindow class, each DocWindow object will have similar characteristics. But, the Name property for each DocWindow object will have a different value. These unique object property values allow you to control objects independently.

Sometimes Word Pro objects are created automatically when the user creates things in the Word Pro user interface, such as frames, tables, and page layouts. You can use LotusScript to access and manipulate these objects just as you do when you create objects yourself.

Word Pro provides two different ways to navigate and view objects in the object model: the Foundry (which allows you to explicitly access a specific object) and Focus (which allows you to access the object in the current context).

Foundry

One means of accessing an object is the Foundry class. Almost every element of a Word Pro document is contained by a collection object within the Foundry object of the division where it was created. The Foundry class has properties that contain instances of most collection classes available in Word Pro. You can access any object in a division by accessing the corresponding collection object in a division's Foundry object.

For example, if you want to change the font of a paragraph style, you can access it by accessing the corresponding ParagraphStyles object, which is a collection. To make the Default paragraph style have bold text, you can use the statement:

```
.Division.Foundry.ParagraphStyles.("Default Text" _
).Font.Bold = True
```

In this statement, the leading dot identifies the Division property of WPApplication (the division where the insertion point is). ParagraphStyles is a collection object that contains a collection of all paragraph styles in the current division. Using parentheses you can index into the collection and find the desired object, in this case the paragraph style named Default Text. Then you can set the Bold property of the Font object to the value True.

Focus

Another means of accessing an object is through focus and the WPApplication object which is instantiated from the WPApplication class. Focus is defined as the location of the insertion point in the document.

When you are using LotusScript in Word Pro, the focus of the insertion point is very important. Focus is usually defined as the place in an application that is currently active. For example, when the insertion point is in Division Two of Document One, Division Two is defined as having the focus. If the insertion point is in Frame Four of Division Two of Document One, Frame Four is defined as having the focus.

The class that represents the actual Word Pro application is WPApplication. The Name property of the object instantiated from the WPApplication class always holds the value Word Pro. Therefore, the WPApplication object is sometimes referred to as the Word Pro object. (It is always called the Word Pro object in the Object drop-down box in the IDE.) Although WPApplication has its own set of properties, methods, and events, it also has current context properties. These are a special set of properties that hold objects that can change depending on the focus. For more information on current context properties, see also “WPApplication” later in this chapter.

Accessing objects that do not have the focus

Although you can use the focus of an object and the Foundry in different ways to access an object, you can also use them together to reach an area of the user interface that does not have the focus.

Using dot notation

To access objects that do not have the focus, most script writers use the leading dot to access the WPApplication object or the predefined global product variable CurrentApplication. For more information on the leading dot, see “Dot notation: using methods and properties with objects” in Chapter 2. However, you can also use the Application property, which is a member of every Word Pro class, to gain access to the WPApplication object.

Note You especially want to use the Application property when a script is written in another programming language or when it is going to be executed by a SmartSuite product other than Word Pro.

Every class in the Word Pro object model has a property named Application. The Application property always has a data type of WPApplication. Therefore, you can use the WPApplication object contained in the Application property of every object to gain access to other parts of the Word Pro application that do not have the focus. To do that, you can traverse the containment hierarchy.

For example, if a block of text has the focus and you want to change the background color of a frame in the next division, you can use the Application property of the corresponding Text object (which contains an instance of the WPApplication class) to access the WPApplication object. Then you can access the frame without changing the focus from the Text object.

The following example uses a ForAll Statement to iterate through all of the Division objects in CurrentDocument, which is a global variable that represents the current TextDocument object. The script uses an If statement to look for the specified division, Body. If the script finds the Body division, it does the following:

- Accesses the internal name of the division
- Indexes into the Divisions collection based on the internal division name
- Indexes into the Frames collection based on the frame name
- Sets the background color of the frame

To use this example in Word Pro, you need to do the following:

1. Create a new document in Word Pro using any SmartMaster.
2. Create two divisions: Body and Division1.

   For more information on creating divisions in a document, search on “Divisions, creating” in the Word Pro Help Index.
3. Create a frame in the Body division.

4. Type text in the Division1 division.
5. Leave the insertion point in the text in Division1.
6. Choose Edit - Script & Macros - Show Script Editor.
7. Type the following script in the space between Sub Main and End Sub.

Note Word Pro divisions have two names: the name that appears on the divider tab and an internal hexadecimal name. Word Pro uses an internal name because two divisions can have the same name on their divider tabs. If you want to access the name that appears on the divider tab (for example, Body, Division 1, and so on), you need to access the Name property of the DivisionInfo object. If you want to access the internal division name, use the Name property of the Division object.

```
Sub Main
   Dim DivName As String
   ForAll MyDivision In CurrentDocument.Divisions
   If MyDivision.DivisionInfo.Name = "Body" Then
      DivName = MyDivision.Name
      With CurrentApplication.Divisions(DivName).Foundry
         .Frames("Frame1" _
            ).Background.BackColor SetRGB 128, 0, 128
         .Color.Override = $LwpColorOverrideRgb
      End With
      Exit Sub
   End If
   End ForAll
End Sub
```

An explanation of the script

Each Division object contains a Foundry object, as explained in the previous section. The Foundry class contains an instance of the FramesLayoutCollection class in the Frames property. The FramesLayoutCollection holds all of the FrameLayout objects in the named division. In order to indicate the frame for which you want to change the background color, you must use a reference (Frame1) to specify the desired FrameLayout object.

For more information on the FrameLayoutCollection class, search on “Classes (LotusScript)” in the Word Pro Help Index.

Each FrameLayout object contains a Background object in the Background property. Each Background object has a BackColor property that contains an instance of the Color class. To set the RGB values for a Color object, you must set the Override property. Using the Override property allows the SetRGB method to override the existing color of the object. For more information on the SetRGB method, search on “Methods (LotusScript)” in the Word Pro LotusScript Help Index.

Using Bind

You can also access an object that does not have the focus by using the Bind keyword. The following uses the Bind keyword to accomplish the same task as the previous example.

```
Sub Bind
   ' Declare a variable for the frame you want to access.
   Dim Frame1 As FrameLayout
   ' Bind the name of the frame to the frame variable.
   Set Frame1 = Bind("!Body:Frame1")
   ' Access the Background property of the FrameLayout object
   ' and change the color.
   With Frame1.Background
        .BackColor.SetRGB 128, 0, 128
        .Color.Override = $LwpColorOverrideRgb
   End With
End Sub
```

For more information on the Bind keyword, search on “LotusScript” in the Word Pro Help Index, then click “LotusScript Index.”

Word Pro predefined global product variables

Word Pro provides several variables that are available any time you are running a script in Word Pro. These predefined global product variables represent instances of three Word Pro classes.

Note Word Pro global product variables are not available when using the Word Pro object model from another programming language or when using LotusScript in another SmartSuite product.

Variable	*Description*
CurrentApplication	Represents the current session of Word Pro and is the object at the top of the containment hierarchy. The variable has the data type WPApplication. CurrentApplication gives you access to the WPApplication class.
CurrentDocument	Represents the current Word Pro document (.LWP file) in the current session of Word Pro (the document in which the insertion point is located). The variable has the data type TextDocument. The same TextDocument object that is represented by this variable is also contained by the ActiveDocument property of the WPApplication object.
CurrentWindow	Represents the window that contains the current document (the window in which the insertion point is located). The variable has the data type DocWindow. The same DocWindow object that is represented by this variable is also contained by the ActiveDocWindow property of the WPApplication object.

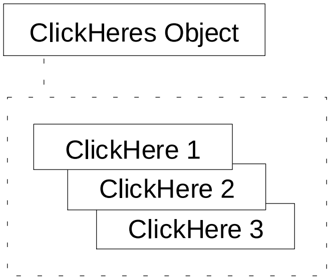
Word Pro collection classes

The Word Pro object model helps you access and enumerate objects by grouping objects of the same type together in collections. For general information about collection classes, see “Collection classes” in Chapter 2. Most of the objects that you need to manipulate in Word Pro have a corresponding collection. For example, the Documents object, which is a collection object, is made up of all objects instantiated from the TextDocument class. Each TextDocument object is an element of the Documents object.

Each collection object has a fixed scope that determines which objects the collection object can hold. Most collection object scopes in Word Pro are limited to the division in which an object is instantiated. For example, the FrameLayoutCollection object holds all the FrameLayout objects for a particular division of the document. Therefore, if the document has three divisions, Word Pro maintains a FrameLayoutCollection object for each division that contains a FrameLayout object.

You can access objects in a collection in two ways:

- Indexing—using the Item method or the indexing syntax to access one specific object in the collection.

  The Item method is a member of every collection class available in Word Pro; however, its parameter is different from class to class. For more information on the Item method, search on “Methods (LotusScript)” in the Word Pro Help Index.

- Iteration—using the ForAll statement to step through the entire collection. For example:

```
ForAll MyStyle in .Division.ParagraphStyles
      Print MyStyle.Name
      MyStyle.Font.Bold = True
End ForAll
```

Common collection classes

The Word Pro object model contains more than 50 collection classes. You will use some of these classes more than others. The following is a list of some of the common collection classes you are likely to use when you write scripts in Word Pro.

Collection class	*Description*
BookMarkCollection	A collection of Bookmark objects in the BookmarkManager class. Use this collection to access any Bookmark object in the BookmarkManager class.
ClickHereCollection	A collection of ClickHere objects in the Foundry of a specific division, document, or application. Use this collection to access all ClickHere objects in a document division.
DivisionCollection	A collection of all Division objects in a Word Pro document. Use this collection to access specific Division objects and Foundry objects.
Documents	A collection of all TextDocument objects in Word Pro. Use this collection to access any TextDocument object in the current session of Word Pro.
DocWindowCollection	A collection of DocWindow objects. Use this collection to access all DocWindow objects that are available during the current session of Word Pro.
FrameLayoutCollection	A collection of FrameLayout objects in a division. Use this collection to access any FrameLayout object in the Foundry object of a specific division.
LayoutCollection	A collection of Layout objects in the Foundry object of a specific division. Use this collection to access any Layout object (such as Header, Footer, and so on) in the Foundry of a specific division.
MarkerCollection	A collection of Marker objects in the Foundry object of a specific division. Use this collection to access any Marker object in the Foundry of a specific division.
ParagraphStyleCollection	A collection of ParagraphStyle objects in the Foundry object of a specific division. Use this collection to manipulate any ParagraphStyle object in the Foundry of a specific division.

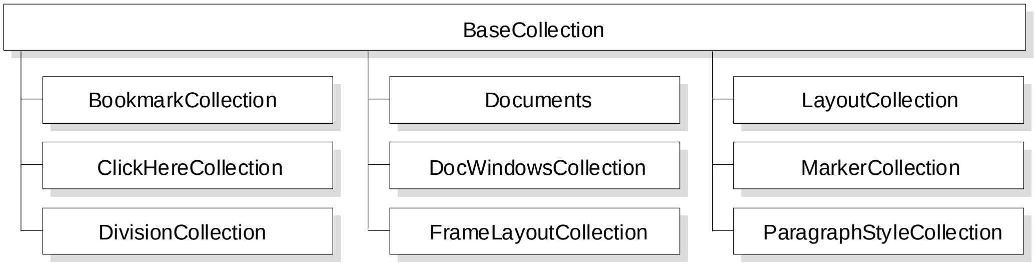
Word Pro abstract classes

The Word Pro object model has several abstract classes. These abstract classes are the fundamentals of the Word Pro object model; familiarity with them helps you understand inheritance. Listed below are some of the more important abstract classes.

BaseCollection

BaseCollection is the class from which all collection classes are derived in Word Pro. The collection classes that are derived from BaseCollection have the following properties: Application, Count, Description, IsValid, Name, Parent, and VersionID. BaseCollection has only one method, IsEmpty, and no events. For more information on collection classes in Word Pro, see “Common collection classes” earlier in this chapter.

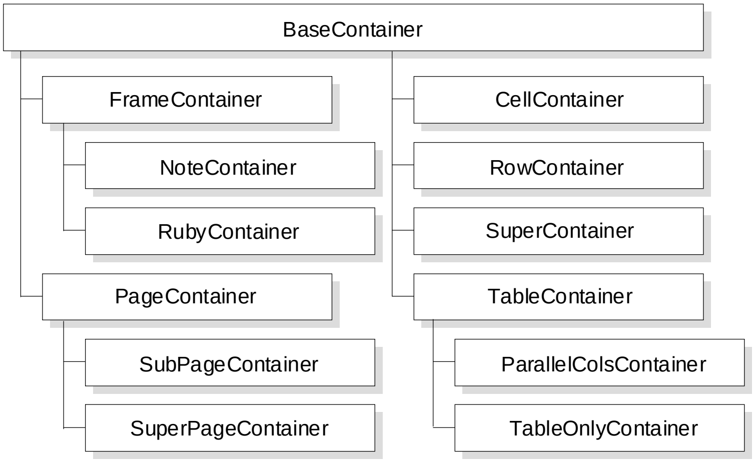
The following is a diagram of some of the important collection classes that are derived from BaseCollection:



BaseContainer

BaseContainer is the abstract class from which all container classes are derived in Word Pro. Container objects are unique to the Word Pro object model. They contain, or hold, similar, related objects that have the focus. For example, if the focus is on a table cell, Word Pro instantiates a container object from the CellContainer class to hold all the objects related to that cell. The CellContainer object comprises a group of related objects, including a CellLayout object and a Cell object, and all the properties, methods, and events that pertain to these objects.

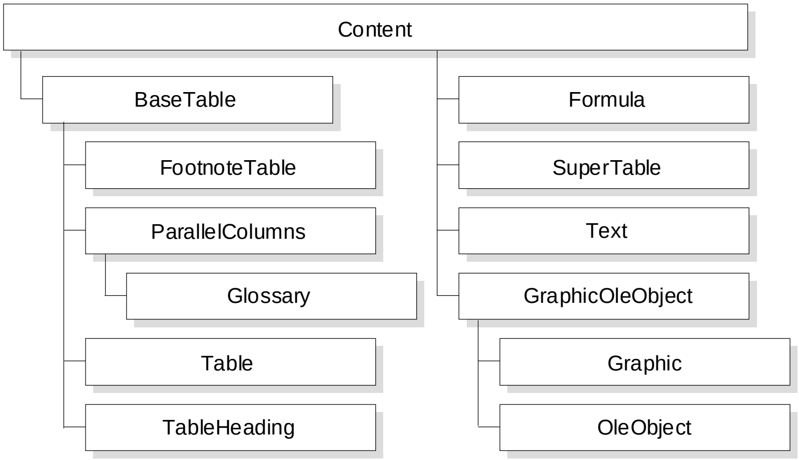
The following is a diagram of the entire inheritance model for BaseContainer. For specific information about the BaseContainer class and containment, see “Other important Word Pro classes” later in this chapter.



Content

Content is the abstract class from which all content classes are derived. The classes derived from Content are used to instantiate objects that represent the contents of a particular type of object. For example, a Formula object represents the content of a CellLayout object; a Text object might represent the contents of a page or the prompt in a Click Here Block; a Graphic object might represent the contents of a frame.

The following is a diagram of the entire inheritance model for Content. For more information about the Content class, search on “Classes (LotusScript)” in the Word Pro Help Index.

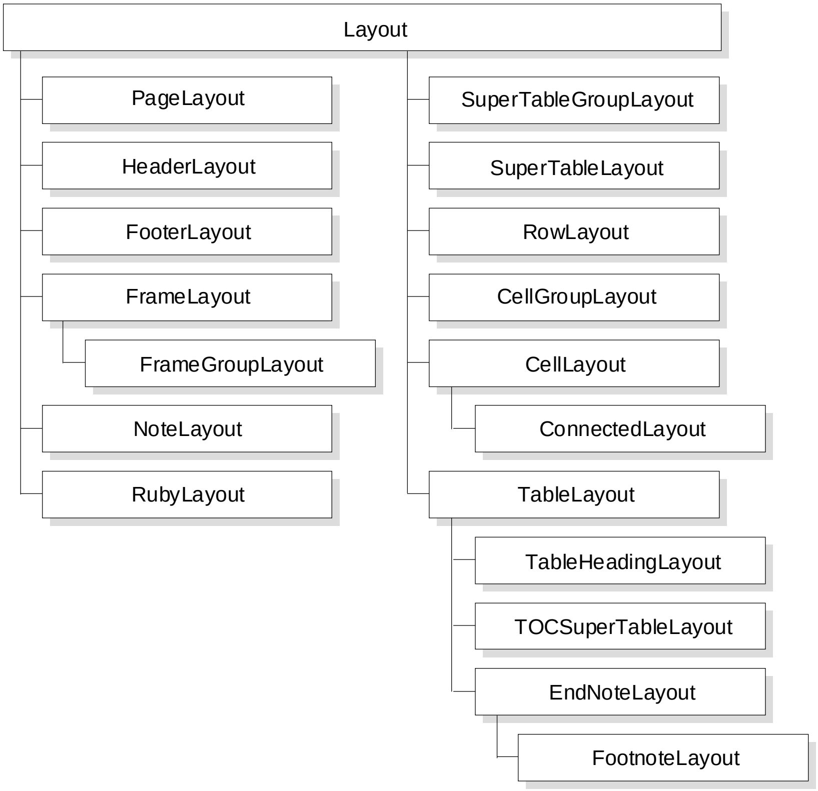


BaseTable

As seen in the diagram above, several classes are derived from BaseTable. Because BaseTable is derived from the Content class, objects instantiated from the classes derived from BaseTable can be used as Content objects.

Layout

While not technically an abstract class, Layout is the base class for all layout classes, including PageLayout, FrameLayout, NoteLayout, TableLayout, RowLayout, and CellLayout. Layout derived classes, such as CellLayout, hold formatting information. For example, CellLayout has properties that deal with the size and position of a cell. Layout derived classes usually contain classes derived from the Content class.



Other important Word Pro classes

When you write LotusScript applications for Word Pro, you should be familiar with several widely used Word Pro classes. Some of these classes are described briefly in this section. For more complete information on these classes and their class members, search on “Classes (LotusScript)” in the Word Pro Help Index.

ClickHere

ClickHereCollection objects hold objects of the type ClickHere. ClickHere objects can be accessed in several ways using the Word Pro containment hierarchy. The most common way of accessing them is through the Foundry object (which is contained by a Division object). When you access the Foundry object of the current division, you can manipulate all the ClickHere objects contained in that particular Division object. For example:

```
.Division.Foundry.ClickHeres.Item("Name")
```

Note The object instantiated from the ClickHereCollection class is the ClickHeres object not the ClickHereCollection object.

BaseContainer

As stated earlier, container objects contain, or hold, similar or related objects that have the focus. They are important because they are an excellent means of accessing Layout objects. For more information on container objects, see “BaseContainer” earlier in this chapter.

Objects instantiated from a container class are temporary and exist only as long as a group of related objects in the user interface has the focus. If you move the insertion point to another object or group of objects in the interface, Word Pro deletes the container object and creates another one for the new group of objects. The parts of a Word Pro document that can have related objects held in container objects include pages, tables, parallel columns, cells, and frames. For example, if a frame has the focus, Word Pro creates a container object that holds all of the objects related to that frame.

The WPApplication class has several properties that can hold container objects. The type of container objects that these properties can hold depends on the location of the insertion point.

For example, if the focus is on a particular table cell that is in a frame on a page, then the container properties of WPApplication would allow you to access the following container objects:

WPApplication property	*Container object*
Cell	Object instantiated from the CellContainer class
Container	Object instantiated from the CellContainer class
Frame	Object instantiated from the FrameContainer class
Page	Object instantiated from the PageContainer class
SuperTableContainer	Object instantiated from the SuperTableContainer class
TableContainer	Object instantiated from the TableContainer class
TableOnlyContainer	Object instantiated from the TableOnlyCont class

Using these properties and the objects they contain, you can gain access to the layouts of any of the objects in the containment hierarchy.

Division

Divisions in Word Pro can hold text, frames, text marked as sections, other divisions with different properties from each other, external files linked to a document, or OLE objects. Therefore, Division objects can contain any of the objects associated with these parts of a division. Each Division object contains an instance of the Foundry class. The Foundry object contains instances of most collection classes that are available in the Word Pro object model. For more information on the Foundry, see “Foundry” earlier in this chapter. For more information on collections, see “Collection classes” in Chapter 2 or “Collection classes in Word Pro” earlier in this chapter.

Documents

The WPApplication class has a Documents property that contains an instance of the Documents class. If you want to access a TextDocument object other than the currently active document, you can use the Documents class.

The following example uses the Documents class to print the names of all open documents:

```
x = 1
ForAll CrntDoc In .Documents
   Print "Doc Number " & x & " - " & CrntDoc.Name
   x = x + 1
End ForAll
```

This example uses a ForAll loop to iterate through the objects collected in the Documents object, since Documents is a collection class. Each time through the loop, the script prints the text string “Doc Number,” the actual number of the document, and the document name. CrntDoc is an object variable for the elements in the Documents object, which is contained in the Documents property of WPApplication. Documents collection elements are always TextDocument objects.

TextCollection

Most text in a document in Word Pro can be represented by a Text object. Text objects are collected in the TextCollection object. You can access the TextCollection object through the Text property of a Foundry object. The Text property of WPApplication provides access to the Text object that currently has the focus. To manipulate a Text object that does not have the focus, you need to know its name. If you do not know its name, you can iterate through the collection that contains the object and find the name.

Sometimes, rather than knowing the name of the Text object, you may know the name of the Layout object that contains it. In this case, the Text object can be accessed by using the Content property of the Layout object. For example, you can use the following syntax to access a Text object that is contained by a FrameLayout object:

```
.Division.Foundry.Frames(FrameObject1).Content.InsertText _
   "Hello"
```

In this last example, you need to know the name of the FrameLayout object that contains the Text object. The Content property of the FrameLayout object gives you access to the Text object.

WPApplication

The WPApplication class is derived from the parent class Application and is crucial to understanding the Word Pro object model. The WPApplication class gives you access to the whole Word Pro application, including the application engine, the Word Pro workspace, and any documents created by the application. By using the members of this class, you can work with any part of Word Pro.

Each time you start Word Pro, a single object is instantiated from the WPApplication class. That object represents the Word Pro application. Unless you run more than one session of Word Pro at a time, only one WPApplication object exists at any given time.

WPApplication has two types of properties—static and current context:

- Static properties, such as Name and UserInterfacePrefs, apply to the Word Pro application as a whole. Their values do not depend on which document or other object is active. The values of static properties remain the same, regardless of which document is active, or where the focus is.
- Current context properties change as the focus moves from one document or division to another. They are called current context properties because their values depend on which objects have the focus.

  For example, in a document with two divisions named ChapterOne and ChapterTwo, you can move the focus from one division to the other. While the focus is on ChapterOne, the current context of the Division object is ChapterOne. Therefore, the Division property of WPApplication contains the ChapterOne Division object. However, when you move the focus of Word Pro to ChapterTwo, the current context changes, and the contents of the Division property change to the ChapterTwo Division object.

The following is a list of some of the more important current context properties and their data types. For a more complete list, search on "Classes (LotusScript)" in the Word Pro Help Index, click "Word Pro Classes," then click "WPApplication."

Current context property	*Data type*
ActiveDocument	TextDocument
ActiveDocWindow	DocWindow
ApplicationWindow	ApplicationWindow
Content	Content
CurrentCell	CellLayout
CurrentColumn	Layout
CurrentRow	RowLayout
Division	Division
Documents	Documents
Foundry	Foundry
Frame	FrameContainer
Graphic	Graphic
Layout	Layout
Page	PageContainer
Table	Table
Text	Text

The global product variable CurrentApplication contains the current WPApplication object.

Using the IDE in Word Pro

You can use the Integrated Development Environment (IDE) in Word Pro to do the following:

- Record a script in the current file
- Record a script as a separate file
- Record a script at the insertion point in the IDE
- Insert a script template
- Access a named object
- Write a sub or function
- Save a script as a module or an .LSS file

To open the IDE in Word Pro, choose Edit - Script & Macros - Show Script Editor. For more information about the IDE, see Chapter 3.

Recording a script

In Word Pro, just as in Ami Pro, you can record your actions in the product and then view the code behind the actions. This is called recording a script. When you record a script, you can insert the code in three places:

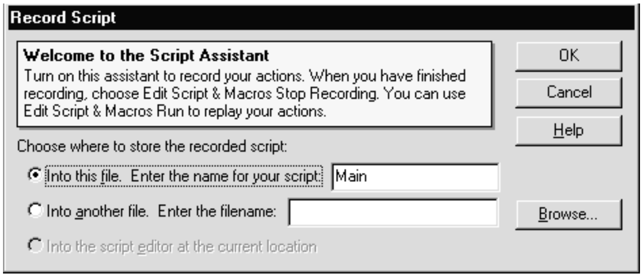
- The current file
- A separate file
- The Script Editor at the current location

Recording a script in the current file or in a separate file

In some cases, you may want to record a script into the current file. For example, if you want to customize a SmartMaster or create startup scripts, you need to save the script as part of the current file. The script is stored as part of the Word Pro (.LWP) or SmartMaster (.MWP) file. In other cases, you may want to record a script into a separate file. Lotus recommends that you save scripts that you want to attach to SmartIcons in a separate file.

To record a script in the current file or another file:

1. Choose Edit - Script & Macros - Record Script.



Welcome to the Script Assistant
Turn on this assistant to record your actions. When you have finished recording, choose Edit Script & Macros Stop Recording. You can use Edit Script & Macros Run to replay your actions.

Choose where to store the recorded script:

● Into this file. Enter the name for your script: Main

○ Into another file. Enter the filename:

○ Into the script editor at the current location

2. Select where you want to store the script:
   - Into this file—You must specify a name for the script. If you do not, Word Pro inserts the recorded script into the Main sub of the !Globals object. If you enter a name for your script, Word Pro inserts the script into a sub of that name associated with the !Globals object.
   - Into another file—You must specify the name of the file in which to place the script. You can select an existing file name by clicking Browse. You can also specify a name for the script by typing a file name, an exclamation point (!) and a name for the sub. For example, type: **`MYFILE.LWP!MYSUB()`**
3. Click OK to start recording.

4. Perform the task or tasks you want to record.
5. After you complete the task, choose Edit - Script & Macros - Stop Recording.

   After you record the script, the IDE opens so you can view and test the script you just recorded.

Recording a script into the Script Editor at the current location

If you are writing a script and want to include the script equivalent of a certain task but do not know what the equivalent statements are, you can record your actions and save the recorded script in the Script Editor. After you record your actions, you can use the Script Editor to view and modify the LotusScript code.

To record a script at the current location:

1. Choose Edit - Script & Macros - Show Script Editor.
2. In the Script Editor, place the insertion point in the script where you want to record the task.
3. In the Script Editor, choose Script - Record at Cursor.
4. Perform a task.

   Note To move the focus back to Word Pro without clicking a specific object, move or minimize the Script Editor and click the Word Pro title bar.
5. After you complete the task, click "Recording" in the Word Pro status bar.

Inserting a script template

As you use LotusScript, you may find yourself entering the same function or other code in numerous scripts. In the Script Editor, Word Pro provides several sets of frequently used scripts as script templates. You can use these templates to insert frequently used functions into the current script.

To insert a script template:

1. In the Script Editor, display the script to which you want to add the script template.
2. Place the insertion point in the script where you want to insert the template.
3. Choose Script - Insert Template.
4. Select a script template.

   For a complete list of script templates available in Word Pro, see the table at the end of this procedure.

5. Click Insert.

   Word Pro places the script or scripts and comments in the current script at the insertion point. For some of the script templates, Word Pro places variable names and remarks in the (Declarations) script for the !Globals object. You can and will probably need to modify the code that was inserted from the script template because the templates contain example text and / or variable names.

For example, in the Cut, Copy, and Paste template below, “Sample text to copy,” will appear in your document when you run the script if you do not modify the template.

```
Sub Main
   ' Copy the selected text.
   .Type("Sample text to copy")
   .Type("[home][ShiftCtrlDown]")
   .CopySelection
   ' Cut the selected text.
   .CutSelection
   ' Paste the selected text.
   .Type("[CtrlDown]")
   .Paste
End Sub
```

The following table lists the script templates available in Word Pro and describes what tasks they accomplish.

Script template	*Description*
Basic - Cut, Copy, and Paste	Selects, cuts (copies), and pastes text.
Basic - Find and Replace	Finds and replaces text.
Basic - List fonts	Builds a list of all available fonts in a parallel column format.
Basic - Set Page margins	Changes margins and page layouts.
Collection - All layouts	Retrieves all of the layouts in the current division and prints the following: layout information, class name, layout name, and editor name.
Collection - All styles	Retrieves all of the paragraph styles in the current division and prints the following: paragraph style name, description, font name, font size, font type, and whether or not the font is bold.
Create a Data Set	Attaches a data set to the current document and names the data set.

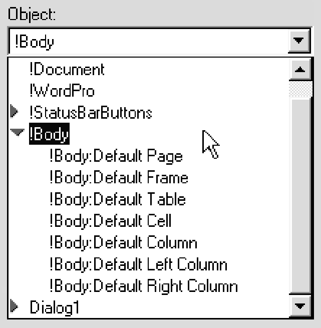
continued

Script template	*Description*
Create a Timer	Creates a named timer within Word Pro, sets the interval in seconds, and turns the timer on.
Display a common dialog box	Creates a common File - Open dialog box for the current operating system.
Frame - modify a frame	Modifies an existing frame layout.
Intermediate - Using Bookmarks	Creates, manipulates, and goes to a bookmark.
Issue a menu/icon command	Issues a menu or icon command to Word Pro.
Menu - Create a new menu item	Creates a new top-level menu item. Also, provides a HitMenu sub to use when a menu item is selected.
Type text	Types text and changes the font. Also, shows how to manipulate the current Text object.

Accessing Word Pro objects and events

Objects appear in the object drop-down box in the IDE because they are capable of raising events in LotusScript. Word Pro objects that raise events automatically appear in the object drop-down box. These objects include the following:

- The !Document object, which represents the current document
- The !Word Pro object, which represents the application
- !StatusBarButtons objects, which include !Font object, !Style object, and so on
- Division objects, such as !Body object, which include the parts of the document, such as !Body:DefaultPage, !Body:DefaultFrame, named layout objects, ClickHere objects, and so on



Note In addition to the objects that raise events, the !Globals object is also listed in the Object drop-down box. The !Globals object is used with scripts that are not tied to a particular Word Pro object.

Although layout objects, such as tables, can raise events, they do not appear automatically in the Object drop-down box. If a layout object does not appear in the Script Editor, it does not have a name. To make a layout object appear in the Script Editor Object drop-down box, you must name it in Word Pro.

For example, if you have a document with a table (which is a layout object) and you want to create a script to handle an event raised by the corresponding table object, you must first name the table.

To name a table:

1. In Word Pro, choose Table - Table Properties.
2. Click the Misc tab.
3. Type a name for the table in the “Name” box.
4. Collapse, move, or close the InfoBox.
5. Choose Edit - Script & Macros - Show Script Editor.

   The object that you named appears in the Object drop-down box in the Script Editor under the division in which it is located.

You can name an object using any name, for example, MyNameClickHere for a Click Here Block. However, when these objects show up in the Object drop-down box, they may appear with a prefix. For example, if the object is part of the body of a document, such as a Click Here Block, the object name appears with the prefix !Body. Therefore, the complete object name for the MyNameClickHere Click Here Block would be !Body:MyNameClickHere.

The objects appear with prefixes in the Object drop-down box because they are contained by other objects. In the example above, the block called MyNameClickHere is contained by the Division object named Body. Every object that is contained by the Division object named Body appears under the heading !Body.

The general format for named objects in Word Pro is as follows:

***DocumentName!TopLevelDivisionName\SecondLevelDivisionName\...
LowestLevelDivisionName:ObjectName***

Sometimes objects are not contained in a Document object and DocumentName is omitted; however, the separator exclamation point (!) is still included. Because of these formatting rules, objects contained in a Division object but not in a Document object appear in the following format in the Script Editor:

!DivisionName:ObjectName

When divisions are not involved in naming objects, the division names and their separators, backslash (\) and colon (:), are also omitted. For example, if a script accesses an object from a collection in the Foundry, the script need only specify ObjectName because the document and division are understood to be the document and division that contain that Foundry object.

When the script accesses the object by using the Bind keyword, however, you must specify the full name of the object including the separators, exclamation point (!), backslash (\), and colon (:), because no document or division is implied when you use the Bind keyword. For more information on the Bind keyword, search on “LotusScript” in the Word Pro Help Index, then click “LotusScript Index.”

Saving scripts

When you are using Word Pro and the IDE, you can save scripts in three different ways.

- You can save scripts with the current document (.LWP file) or SmartMaster (.MWP file).
- You can export scripts for the !Globals object of a compiled script to a LotusScript Object (.LSO) file.
- You can export the script source code as a plain text LotusScript Script (.LSS) file.

Saving a script with the current file

If your script needs to execute each time a new document is created from a particular SmartMaster, you will need to save the script as part of a SmartMaster file (.MWP). If your script is designed to work only with a particular document, you should save your script as part of that Word Pro document file (.LWP).

Note Scripts that are designed to work with multiple SmartSuite products can also be stored in an .LWP or .MWP file. However, such scripts can only be executed from Word Pro.

To save a script as part of an .MWP or an .LWP file:

1. Open the file.
2. Choose Edit - Script & Macros.
3. Choose Show Script Editor or Record Script.
4. Enter your script in the Script Editor by typing it in or by recording it.
5. In the IDE, choose File - Save Scripts.

   The IDE saves the script as part of the current file (.MWP or .LWP). Every time that you open this file, you can open the Script Editor to view and edit the script.

   To edit a script that is part of a Word Pro SmartMaster, you must open the .MWP file.

Saving a script as an .LSO file

If you need to execute your script within multiple SmartSuite products, you can save the object code as an .LSO file. In the IDE, choose File - Export Globals As LSO. The IDE saves the code as an .LSO file in the specified directory.

Note When you export scripts to an .LSO file, the IDE only exports compiled code that is part of the !Globals object.

If you want to call a sub or function in an .LSO file, use the following syntax:

```
Use filename.lso
MyProcedure
```

Because an .LSO file contains object code, you cannot edit it directly. To change an .LSO file, you must edit the source code contained in the corresponding .LWP, .MWP, or .LSS file.

Saving a script as an .LSS file

Sometimes you may want to save your script as an external text file. In this case, you can export your script to an .LSS file. In the IDE, choose File - Export Script. The IDE saves the current script or all the scripts for an object in the current document as an .LSS file in the specified directory.

If you want to call a sub or function in an .LSS file, use the following syntax to include the .LSS file:

```
%Include "filename.lso"
```

Using the Dialog Editor in Word Pro

To open the Dialog Editor in Word Pro, choose Edit - Script & Macros - Show Dialog Editor. Word Pro saves dialog boxes that you create using the Dialog Editor with the Word Pro file (.LWP) in which you created them.

For more information about using the Dialog Editor, see “Developing custom dialog boxes in the Dialog Editor” in Chapter 3.

Migration information

The two types of macros that you can create in Ami Pro are recorded macros and coded macros. If the macro was recorded in Ami Pro, you must open the macro inside Ami Pro and save it as a macro file (.SMM) before you try to run it in Word Pro. The process of opening the macro and saving it as an .SMM file places the recorded macro in a Sub statement and makes it easier for Word Pro to convert the macro. If a macro is coded using the Ami Pro macro language (that is, if it is part of a statement with Function and End Function lines), you can run it in Word Pro, and Word Pro automatically converts the macro to a format that LotusScript can read.

Note If you attempt to play an Ami Pro recorded macro in Word Pro without first displaying and saving it in Ami Pro, Word Pro displays an error message.

Saving a recorded Ami Pro macro before conversion

To save an Ami Pro macro as an .SMM file:

1. Open Ami Pro.
2. Choose File - Open.
3. Choose Ami Pro Macro in the “List files of type” box.
4. Specify the name of the macro you want to convert in the “File name” box.
5. Click OK.

   Ami Pro displays the macro in the document window.
6. Choose File - Save.
7. Choose File - Close.

Converting an Ami Pro macro

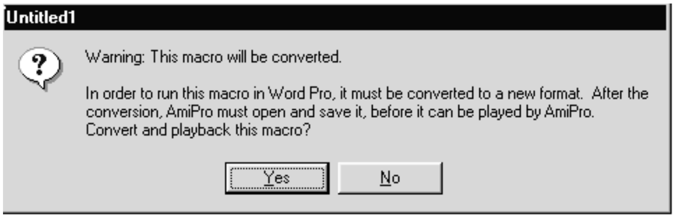
Although it is possible to run most Ami Pro 3.x macros inside Word Pro 97, Word Pro must first convert the macro to a file that LotusScript can read—a type of hybrid ASCII file.

Note Lotus recommends that you back up the macro to a separate file before you run it inside Word Pro.

To convert an Ami Pro macro:

1. In Word Pro, choose Edit - Scripts & Macros - Run.
2. In the Run Script dialog box, select "Run script saved in another file."
3. Specify the name of the Ami Pro macro that you want to run in Word Pro.
4. Click OK.

   Word Pro displays a message before it converts the macro.



Untitled1

Warning: This macro will be converted.

In order to run this macro in Word Pro, it must be converted to a new format. After the conversion, AmiPro must open and save it, before it can be played by AmiPro.
Convert and playback this macro?

5. Click Yes.

   Word Pro runs the macro.

Running existing macros

After you convert a macro, you still may not be able to successfully run it. If a macro will not run or will not run properly inside Word Pro, the whole macro or one of its functions may have a compatibility problem. Word Pro may not be able to run a macro for any of the following reasons:

- The function is not supported in Word Pro.

  Some Ami Pro functions are not supported in Word Pro at all. For example, the Ami Pro functions ShowStylesBox, HideStylesBox, and ToggleStylesBox are not supported in Word Pro because Word Pro does not use the Styles Box.

- The function is supported in a different manner in Word Pro.

  Most of the Word Pro user interface and functionality is completely different from the Ami Pro user interface and functionality. This difference can lead to macro incompatibility.

The Type function is a good example of how the user interface and functionality have changed. The Type function is still fully supported in that all keystrokes specified to be typed are still sent to the document to be typed; however, Word Pro responds differently to some keystrokes. For example, in Ami Pro, function keys are used to select a paragraph style, but in Word Pro, function keys are used as CycleKeys. If the Ami Pro macro selects styles using the Type function, it will not have the same functionality in Word Pro.

- Options or parameters of the function are not supported in Word Pro.

  In some cases, a macro function is supported, but because of product differences, one or more of the function options or parameters are not supported. For example, the New function, which creates a new document, is supported in both Ami Pro and Word Pro.

  However, in Ami Pro, options for the New function include bringing in the contents of a style sheet. In Word Pro, the content of style sheets (SmartMaster templates) are always brought in.

- The function is supported in Word Pro, but cannot be converted.

  Occasionally, functionality that is supported by both Ami Pro and Word Pro is not supported by the macro conversion process. Examples of product functionality that may not be converted include master document, table of contents, and index. Because the same functionality can be very different in the two products, some required parameters for Word Pro cannot be supplied using the Ami Pro macro language. Therefore, the macro does not convert correctly.

  Note When you try to run functions that are not convertible, Word Pro displays a message that indicates that the function is not supported.

- The function uses 16-bit API calls.

  Some macros may contain Windows 3.1 API calls. Since Word Pro 97 is a 32-bit, or Windows 95, product, it cannot convert or use 16-bit API calls for two reasons. First, all handles in Windows 95 moved from 16-bit to 32-bit, thus changing the signature for all Windows calls. Second, the way you call a 16-bit DLL differs from the way you call a 32-bit DLL. Therefore, Word Pro cannot execute calls to 16-bit DLLs in macros that contain them. For example, if your macro contains DllLoad or DllCall functions, the macro will not run in Word Pro.

Strategies for editing Ami Pro macros

If your Ami Pro 3.x macro does not run in Word Pro, open the macro in Word Pro and try one or more of the following:

- Remove any nonsupported functions.
- Remove any 16-bit API calls.
- Verify that all macro parameters and values have equivalent parameters and values in Word Pro.

If you try to run a macro that creates and formats documents and it does not execute, you can use a Word Pro SmartMaster as a substitute. Because SmartMaster templates can contain Click Here Blocks and because they have the ability to include multiple page layouts, they can replace much of the functionality of this type of macro.

If you try to run a macro that was recorded in Ami Pro and it does not execute, you can rerecord the functionality using the IDE. A recorded script will play back faster than an Ami Pro 3.x macro, and you will not have to spend time converting it.

Team Computing in Word Pro

The diverse and powerful Team Computing features of Word Pro can be accessed using LotusScript and Word Pro objects. You can write scripts to modify the following:

- Editor access
- Markup options
- Document access
- Editing rights

Note You cannot use LotusScript to bypass Team Computing restrictions. For example, Word Pro will never allow an unauthorized editor to access a document that is password-protected with TeamSecurity, even using LotusScript.

The following are examples of scripts that will enhance your Team Computing capabilities.

Modifying editor access

TeamReview and TeamSecurity are Word Pro Team Computing features that allow the user to determine who can review a document, how much control a reviewer has, and how the document can be distributed. To access TeamReview and TeamSecurity and use their features, the user needs to set options in the TeamReview Assistant and in the TeamSecurity Assistant.

The ReviewRightsExample script sets the following TeamSecurity and TeamReview options:

- A standard greeting for all editors of the document
- All editors' access rights as read-only, other than for the current editor
- The current editor's access rights as unlimited

To run this example, you can type the following script (described in detail in the next section) in the space between Sub Main and End Sub.

Note The text of this script is stored in DW10_S1.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
Option Public
%Include "WPBITMSK.LSS"

Sub ReviewRightsExample
' Get name of the current editor and set it equal to the
' variable CrntEditor. Then, set up a greeting for the
' document.
Dim MyGreeting As String
Dim CrntEditor As String
CrntEditor = CurrentApplication.Preferences.UserName
MyGreeting = InputBox("What greeting do you want to" _
   +" display?","Set Review Option")
' Set TeamReview properties to prevent anyone other than
' the current editor from editing this document.

   With .ActiveDocument

      .DocControl.UseGreeting = True
      .DocControl.Greeting = MyGreeting
      .DocControl.FileProtectionType = _
        $LwpFileProtectTypeEditors
      .EditorManager.Editors("All Others").Abilities = _
        $LwpEditAbilEditingNotAllowed
      .EditorManager.Editors("All Others").Locks = _
        LwpEditLocksNoCopyAndNoSaveas
      .EditorManager.Editors("All Others").Suggestions = _
        LwpEditSuggEditingInNewVersion
```

```
     .EditorManager.Editors(CrntEditor).Abilities = _
       $LwpEditAbilEditCurrentOrNewVer
     .EditorManager.Editors(CrntEditor).Locks = _
       LwpEditLocksNoLocks
     .EditorManager.Editors(CrntEditor).Suggestions = _
       LwpEditSuggNoSuggestions
   End With
End Sub
```

An explanation of the script

The first section of the script uses the %Include directive to load the WPBITMSK.LSS file. WPBITMSK.LSS is located in the main Word Pro directory, and it contains a list of constants that you can use as values for properties. When you write scripts for Team Computing features, you may need to use WPBITMSK.LSS to set properties.

The first two lines of the sub ReviewRightsExample declare two variables, CrntEditor and MyGreeting.

Next, the script assigns values to the two variables. CrntEditor stores a string that contains the name of the current editor.

```
CrntEditor = CurrentApplication.Preferences.UserName
```

CurrentApplication, which represents the current Word Pro application, has a property named Preference which contains an instance of the Preferences class. The Preferences class has UserName as a property. CrntEditor stores the name that is stored in the UserName property.

MyGreeting stores a string of text that the user enters in an input box.

```
MyGreeting = InputBox("What greeting do you want" _
   +" to display?","Set Review Option")
```

To assign MyGreeting a value, LotusScript displays an input box with the text “What greeting do you want to display?” and provides a text box for the user’s response. “Set Review Option” is the title of the input box.

This script uses a With statement to access the properties of the TextDocument class. The DocControl property and the EditorManager property provide access to the DocControl object and the EditorManager object.

```
.DocControl.UseGreeting = True
.DocControl.Greeting = MyGreeting
.DocControl.FileProtectionType = $LwpFileProtectTypeEditors
```

The DocControl class allows you to access a document, assign editing rights, enable password protection, select or change colors that show editor markups, make insertions and deletions, and enable document protection in a division. This example sets the following options in the TeamSecurity Assistant using properties of the DocControl object:

Property	*TeamSecurity Assistant option*	*Value*
UseGreeting	Editing Rights tab - “Display Greeting with this text” check box	Checked.
Greeting	Editing Rights tab - “Display Greeting with this text” text box	Sets the greeting in the dialog box to the greeting the user entered in the input box.
FileProtectionType	Access tab - “Who can open (access) this document” group box	Selects the “Current editors only” button. Only those listed as editors on the Editing Rights panel can open the document.

This portion of the script sets options in the TeamReview Assistant.

```
.EditorManager.Editors("All Others").Abilities = _
 $LwpEditAbilEditingNotAllowed
.EditorManager.Editors("All Others").Locks = _
  LwpEditLocksNoCopyAndNoSaveas
.EditorManager.Editors("All Others").Suggestions = _
  LwpEditSuggEditingInNewVersion
.EditorManager.Editors(CrntEditor).Abilities = _
 $LwpEditAbilEditCurrentOrNewVer
.EditorManager.Editors(CrntEditor).Locks = _
  LwpEditLocksNoLocks
.EditorManager.Editors(CrntEditor).Suggestions = _
  LwpEditSuggNoSuggestions
```

Editors is a property of the EditorManager class, and it holds an instance of the EditorCollection class. By using the Editors property of the EditorManager class to index into the EditorsCollection object, the script indicates which editors it is setting options for in the TeamReview Assistant. To do so, it uses the String value “All Others” and the variable CrntEditor. “All Others” is a literal value in the EditorsCollection that represents all editors other than the current editor and the SmartMaster that created the document. CrntEditor is the variable that holds the value of the UserName property. The script uses properties of the Editor class (the EditorsCollection class is a collection of Editor objects) to set options in the TeamReview Assistant.

The following is a list of options that the script sets for “All Others”:

Property	*TeamReview Assistant option*	*Value*
Abilities	Step 2: What tab - “Edits are” drop-down box	Not allowed (read-only)
Locks	Step 2: What tab - “Limited to” drop-down box	No copying or saving as a new file
Suggestions	Step 2: What tab - “Greeting will suggest” drop-down box	Editing in a new version

The following is a list of options that the script sets for the current editor:

Property	*TeamReview Assistant option*	*Value*
Abilities	Step 2: What tab - “Edits are” drop-down box	Allowed in current version or new version
Locks	Step 2: What tab - “Limited to” drop-down box	(No limits)
Suggestions	Step 2: What tab - “Greeting will suggest” drop-down box	(Nothing)

This script can run when a menu option is chosen or when the user clicks a custom icon. For more information on running a script, see “Running the Memo Signing Script” in Chapter 4.

Modifying markup options

Sometimes when a user receives a document, he or she may want to use Review & Comment Tools to mark the document for editing. In the TeamSecurity Assistant, users can set markup options for themselves, or if they have the access rights, for anyone else reviewing the document.

The MarkupOptions example below allows the user to set TeamSecurity markup options without having to view the Markup Options dialog box. It also sets access options for the current editor.

To run this example, you can insert the following script in the space between Sub Main and End Sub.

Note The text of this script is stored in DW10_S2.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
Option Public
%Include "WPBITMSK.LSS"
Sub MarkupOptions
   Dim CrntEditor As String
   CrntEditor = CurrentApplication.Preferences.UserName
   ' Section 1 - Specify markup options for the
   ' current editor.
   With .ActiveDocument.EditorManager

   ' Set font and attributes for inserted text by first
   ' making sure the text reverts to its original style
   ' then changing the color of the text to purple.

   .Editors(CrntEditor).InsertFont.FontColor.Red = 255
   .Editors(CrntEditor).InsertFont.FontColor.Blue = 255
   .Editors(CrntEditor).InsertFont.FontColor.Green = 0
   .Editors(CrntEditor).InsertFont.FontColor.Override = _
      $LwpColorOverrideRgb
   .Editors(CrntEditor).InsertFont.DoubleUnderline = False
   .Editors(CrntEditor).InsertFont.Underline = False
   .Editors(CrntEditor).InsertFont.Bold = False
   .Editors(CrntEditor).InsertFont.Italic = True

   ' Section 2 - Specify font and attributes for
   ' deleted text.
   .Editors(CrntEditor).DeleteFont.FontColor.Red = 128
   .Editors(CrntEditor).DeleteFont.FontColor.Blue = 0
   .Editors(CrntEditor).DeleteFont.FontColor.Green = 0
   .Editors(CrntEditor).DeleteFont.FontColor.Override = _
      $LwpColorOverrideRgb
   .Editors(CrntEditor).DeleteFont.Overstrike = True

   ' Section 3 - Specify a color for highlighting text.
   .Editors(CrntEditor).HiLiteColor.Red = 128
   .Editors(CrntEditor).HiLiteColor.Blue = 128
   .Editors(CrntEditor).HiLiteColor.Green = 255
   .Editors(CrntEditor).HiLiteColor.Override = _
      $LwpColorOverrideRgb

   ' Section 4 - Specify access rights so the current
   ' editor has unlimited access.
   .Editors(CrntEditor).Abilities = _
      $LwpEditAbilEditNewVersionsOnly
   .Editors(CrntEditor).Locks = LwpEditLocksNoLocks
   .Editors(CrntEditor).Suggestions = _
      LwpEditSuggNoSuggestions
   End With

End Sub
```

An explanation of the script

The first section of the script uses the %Include directive to load the WPBITMSK.LSS file. WPBITMSK.LSS contains a list of constants that you can use as values for properties, and it is located in the main Word Pro directory. When you write scripts for Team Computing features, you may need to use WPBITMSK.LSS to set properties.

The first line of the sub MarkupOptions declares a variable, CrntEditor. The script assigns a value to the variable. It stores a string that contains the name of the current editor.

```
CrntEditor = CurrentApplication.Preferences.UserName
```

CurrentApplication, which represents the current Word Pro application, has a property named Preferences, which contains an instance of the Preferences class. The Preferences class has UserName as a property. CrntEditor stores the name that is stored in the UserName property.

This script uses a With statement to access the properties of the TextDocument class. The TextDocument class contains an instance of the EditorManager class, which has an Editors property. The Editors property contains an instance of the EditorsCollection class. The EditorsCollection is a collection of all the Editor objects. The InsertFont and DeleteFont properties (of the Editor class) contain instances of the Font class, which represent the font attributes to be applied to inserted and deleted text. By using the EditorManager object to index into the EditorCollection object, the script indicates which editors it is setting options for in the Markup Options dialog box.

In section one of the example, the With statement sets the markup options, such as font color and italics, for inserted text in a TeamSecurity session.

```
.Editors(CrntEditor).InsertFont.FontColor.Red = 255
.Editors(CrntEditor).InsertFont.FontColor.Blue = 255
.Editors(CrntEditor).InsertFont.FontColor.Green = 0
.Editors(CrntEditor).InsertFont.FontColor.Override = _
   $LwpColorOverrideRgb
```

The first three lines of code above set the color of the font for text inserted during a TeamReview session. The combined value of these three color properties, Red, Green, and Blue, is equivalent to the color that a user would choose in the “Markup for insertions: Text color” drop-down box in the Markup Options dialog box. The last line sets the Override property of the Color object (which is contained by FontColor) to $LwpColorOverrideRgb. You can also use the enumerated value 2016 to substitute for $LwpColorOverrideRgb. When the Override property value is $LwpColorOverrideRgb, the Color object color is defined by the values in the Red, Green, and Blue properties.

```
.Editors(CrntEditor).InsertFont.DoubleUnderline = False
.Editors(CrntEditor).InsertFont.Underline = False
.Editors(CrntEditor).InsertFont.Bold = False
.Editors(CrntEditor).InsertFont.Italic = True
```

The last lines of the first section set properties of the InsertFont object. These statements turn off double underlining, underlining, and bold by assigning the DoubleUnderline, Underline, and Bold properties the value False. The last statement turns on italics by assigning the value True to the Italic property.

Section two of the example sets the markup options for deleted text in a TeamSecurity session.

```
.Editors(CrntEditor).DeleteFont.FontColor.Red = 128
.Editors(CrntEditor).DeleteFont.FontColor.Blue = 0
.Editors(CrntEditor).DeleteFont.FontColor.Green = 0
.Editors(CrntEditor).DeleteFont.FontColor.Override = _
   $LwpColorOverrideRgb
```

Just as in the first section, the example assigns values to the Red, Blue, and Green properties of the Color object to create a color for the text. The combined value of these three color properties is equivalent to the color that a user would select in the “Markup for deletions: Text color” drop-down box in the Markup Options dialog box. This section also assigns the $LwpColorOverrideRgb value to the Override property of the Color object.

```
.Editors(CrntEditor).DeleteFont.Overstrike = True
```

The last line of the second section assigns a value to the Overstrike property of the DeleteFont object. By assigning the property the value True, the script turns on overstriking. This value is the same as selecting Overstrike in the “Markup for deletions” drop-down box in the Markup Options dialog box.

Section three of the example sets the markup options for highlighted text in a TeamSecurity session.

```
.Editors(CrntEditor).HiLiteColor.Red = 128
.Editors(CrntEditor).HiLiteColor.Blue = 128
.Editors(CrntEditor).HiLiteColor.Green = 255
.Editors(CrntEditor).HiLiteColor.Override = _
   $LwpColorOverrideRgb
```

Just as in the first two sections, the example assigns values to the Red, Blue and Green properties of the Color object. However, this Color object is contained by the HiLiteColor object, not the FontColor object, because these properties are assigning a value for highlighted text.

Section four of the example sets access rights in the TeamSecurity Assistant for the current editor.

```
' Section 4 - Specify access rights so the current editor has
' unlimited access.
.Editors(CrntEditor).Abilities = _
  $LwpEditAbilEditCurrentOrNewVer
.Editors(CrntEditor).Locks = LwpEditLocksNoLocks
.Editors(CrntEditor).Suggestions = LwpEditSuggNoSuggestions
```

The script uses properties of the Editor class to set the following options for the current editor:

Property	*TeamReview Assistant option*	*Value*
Abilities	Step 2: What tab - "Edits are" drop-down box	Allowed in current version or new version
Locks	Step 2: What tab - "Limited to" drop-down box	(No limits)
Suggestions	Step 2: What tab - "Greeting will suggest" drop-down box	(Nothing)

This script can run when a menu item is chosen or when the user clicks a custom icon. For more information on running a script, see "Running The Memo Signing Script" in Chapter 4.

Modifying document access

TeamSecurity features in Word Pro allow users to limit document access to themselves or to a restricted number of editors. You can write a script that limits access to a document to only the author of that document.

To run this example, you can add the following script in the space between Sub Main and End Sub.

Note The text of this script is stored in DW10_S3.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
Sub DocAccess

   ' Declare the variable CrntEditor and assign it the value
   ' of the current user's name.
   Dim CrntEditor As String
   CrntEditor = .Preferences.UserName

   ' Restrict editing of this document to the current editor.
   CurrentDocument.DocControl.DocControlRestrictedToEditor _
      = CrntEditor

      CurrentDocument.DocControl.FileProtectionType = _
         $LwpFileProtectTypeOrigAuthor

End Sub
```

An explanation of the script

This example uses the DocControl object that is contained by the CurrentDocument global product variable to assign a value to two properties: DocControlRestrictedToEditor and FileProtectionType. CurrentDocument, which represents the current text document, is a predefined Word Pro global variable. CurrentDocument contains an instance of the TextDocument class, and the TextDocument class has a property named DocControl, which contains an instance of the DocControl class. DocControlRestrictedToEditor is a property of the DocControl class that has a data type of String. This script assigns the value of the variable CrntEditor, which represents the current user, to this property.

The example also sets the property FileProtectionType to $LwpFileProtectTypeOrigAuthor. FileProtectionType is a property of the DocControl class. It has a Variant data type and can have various values, such as original author, anyone, or only certain specified editors. In this script, FileProtectionType is set to the enumerated constant for the original author. (You can use the numeric equivalent 262 instead of the enumerated constant $LwpFileProtectTypeOrigAuthor.) This value indicates that the only person who can edit this document is the original author.

This script can run when a menu item is selected or when the user clicks a custom icon. For more information on running a script, see “Running The Memo Signing Script” in Chapter 4.

Note If the user is not the original author or if the user does not have access to the TeamSecurity Assistant for this document, this script will cause an error when the user attempts to run it.

Modifying editing rights

The TeamSecurity features in Word Pro also allow the user to control who has access to the TeamSecurity Assistant and in what version an editor can view a particular file. The TeamSecurity EditingRights script does the following:

- Restricts access to the TeamSecurity Assistant to the current editor
- Sets access to the document
- Restricts all editors to making remarks in new versions of the document
- Allows the current editor to edit the document with no restrictions

To run this example, you can insert the following script in the space between Sub Main and End Sub.

Note The text of this script is stored in DW10_S4.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
Option Public
%Include "WPBITMSK.LSS"

Sub EditingRights
   ' Declare the variable CrntEditor and assign it the value
   ' of the current user's name.
   Dim CrntEditor As String
   CrntEditor = CurrentApplication.Preferences.UserName

   With .ActiveDocument

   ' Restrict access to the TeamSecurity Assistant to the
   ' current editor.
   .DocControl.DocControlRestrictedToEditor = CrntEditor

   ' Specify who has access to open the document.
   .DocControl.FileProtectionType = $LwpFileProtectTypeNone

   ' Restrict all editors other than the current editor to a
   ' new version with remarks only, and do not allow them to
   ' make any suggestions.
   .EditorManager.Editors("All Others").Abilities = _
      $LwpEditAbilEditNewVersionsOnly
   .EditorManager.Editors("All Others").Locks = _
      LwpEditLocksRevmarkOnly
   .EditorManager.Editors("All Others").Suggestions = _
      LwpEditSuggNoSuggestions

   ' Specify the current editor's rights as unlimited.
   .EditorManager.Editors(CrntEditor).Abilities = _
      $LwpEditAbilEditCurrentOrNewVer
   .EditorManager.Editors(CrntEditor).Locks = _
      LwpEditLocksNoLocks
   .EditorManager.Editors(CrntEditor).Suggestions = _
      LwpEditSuggNoSuggestions
   End With

End Sub
```

An explanation of the script

The first section of the script uses the %Include directive to load the WPBITMSK.LSS file. WPBITMSK.LSS is located in the main Word Pro directory, and it contains a list of constants that you can use as values for properties. When you write scripts for Team Computing features, you may need to use WPBITMSK.LSS to set properties.

This script uses a With statement to access the properties of the TextDocument class. The DocControl property and the EditorManager property provide access to the DocControl object and the EditorManager object. For more information about the DocControl object and the EditorManager object, see “Modifying editor access” earlier in this chapter.

This example sets the following options in the TeamSecurity Assistant using properties of the DocControl Object:

Property	*TeamSecurity Assistant option*	*Value*
DocControlRestrictedTo-Editor	Access tab - “Who can open this dialog, and change access, editing rights, and other protection options“ option buttons	Selects “Only.” The name in the drop-down box is assigned the value of the variable CrntEditor.
FileProtectionType	Access tab - “Who can open (access) this document” option buttons	Selects “Anyone (unprotected).” Anyone can open the document.

By using the Editors property of the EditorManager class to index into the EditorsCollection object, the script indicates which editors it is setting options for in the TeamReview Assistant. This example sets the same options as in the editor access example. For more information about the options and their values, see “Modifying editor access” earlier in this chapter.

Top tasks

This section describes several common Word Pro tasks and illustrates how you can use LotusScript to automate the processes associated with them. All of the code is available online in the sample files directory, so you can copy and paste it into the IDE. The name of the example file is listed with each individual example.

Automating a SmartMaster

The most common use of a word processor is to type and format text, usually in a letter, a fax, or a report. Word Pro uses SmartMaster templates and Click Here Blocks to automate typing and formatting text. When you create a Word Pro SmartMaster, you can create Click Here Blocks that prompt users for specific information.

The following script takes a SmartMaster and multiple Click Here Blocks and uses them to automate the process of gathering and inputting information. With this script, you can ask the user specific questions and then insert the responses automatically into the appropriate Click Here Blocks in the document.

The subs in the following example are set to run when a user creates a document using the SmartMaster DW10_S5.MWP. In other words, the subs in the example can form a startup script that runs when the Created event of a document occurs. This example uses the Click Here Blocks that are available in the DW10_S5.MWP SmartMaster.

To run this example, you can open the DW10_S5.MWP SmartMaster that is available online in the sample files directory, or you can enter the following script into the appropriate event sub. You can create a dialog box with buttons to attach this sub to or use the one that is available in the SmartMaster.

```
Sub Created(Source As TextDocument, DocName As String)
   ' Created event script for the document
   DataDialog.Show
End Sub

Sub Load(Source As LotusDialog)
   ' This sub is the Load event script for the dialog box
   ' DataDialog. It clears all of the text boxes and sets the
   ' focus in the first text box.
   Source.RecName.Caption = ""
   Source.Phone.Caption = ""
   Source.Subject.Caption = ""
   Source.RecName.SetFocus
End Sub

Sub Click(Source As LotusCommandButton)
   ' This is the Click event script for the OK button
   ' cmdOK. This sub hides the dialog box while the Click
   ' Here Blocks are being populated.
   Dim TxtRecName As String
   Dim TxtPhone As String
   Dim TxtSubject As String

   With Source.Parent
      .Hide

      ' Get the user input from dialog box.
      TxtRecName = .RecName.Text
      TxtPhone = .Phone.Text
      TxtSubject = .Subject.Text

      'Insert the user's responses into Click Here Blocks.
      With CurrentApplication.Foundry.ClickHeres
         .Item("RecipientName").DeleteContents
         .Item("RecipientFaxNumber").DeleteContents
         .Item("SubjectLine").DeleteContents
         .Item("RecipientName").InsertText TxtRecName
```

```
            .Item("RecipientFaxNumber").InsertText TxtPhone
            .Item("SubjectLine").InsertText TxtSubject
         End With

         ' Close the dialog box
            .Close
      End With
   End Sub

   Sub Click(Source As LotusCommandButton)
      ' This is the Click event script for the Cancel button
      ' cmdCancel. It closes the dialog box if the user clicks
      ' Cancel.
      Source.Parent.Close
   End Sub
```

Validating Click Here Blocks

Sometimes when you create a Click Here Block in a SmartMaster, you want to make sure that the person who uses the Click Here Block fills in the exact type of information that is requested. For example, you may want the user to fill in a Click Here Block with a Social Security number composed of a nine-digit number with dashes.

Word Pro ClickHere objects allow you to write scripts that validate the type of information that they contain. The following example uses the ExitClickHere event for the object named ClickHere31 to make sure that the user entry is either a nine-character numeric response or an eleven-character entry that contains dashes. If the entry is neither, the script displays a message box when the user exits the Click Here Block.

To run this example, you can open the DW10_S6.MWP SmartMaster that is available online in the sample files directory, or you can insert the following into the appropriate event sub.

```
Sub ExitClickHere(Source As ClickHere, _
   ClickHereName As String)

   ' Define three constants: a message box, a number length,
   ' and a number length that includes dashes.
   Const MsgTxt = "Contents is not correct format, please" _
      +" enter a valid SSN."
   Const NumLgth = 9
   Const DashLgth = 11

   ' Define a variable for the contents of the
   ' Click Here Block.
   Dim MyContents As String

   ' Retrieve the contents of the Click Here Block
   MyContents = Source.GetMarkedText
   ' Check the user response in the Click Here Block to see
   ' if it has 11 characters (DashLgth) and that the fourth
   ' and seventh characters are dashes.

   If Len(MyContents) = DashLgth And Mid$(MyContents,4,1) = _
      "-" And Mid$(MyContents,7,1) = "-" Then

      ' If it has 11 characters, make sure that all
      ' characters that the user entered between the dashes
      ' are numeric.
      If Not(IsNumeric(Mid(MyContents,1,3)) And _
         IsNumeric(Mid(MyContents,5,2)) And _
         IsNumeric(Mid(MyContents,8,4))) Then

      ' If the characters entered between dashes are not
      ' numeric, display a message box with the value of
      ' the String constant MsgTxt to ask the user
      ' to enter a valid Social Security number.
      MessageBox MsgTxt

         ' Go back to the Click Here Block and select the
         ' user's previous response.
         Source.GoTo(True)
      End If

   ' Otherwise, if the user entry is numeric and if it is
   ' 9 characters long (NumLgth), format the numbers as a
   ' Social Security number with dashes. Delete the user
   ' entry and insert the new formatted number (MyContents)
   ' into the Click Here Block.

  ElseIf
   IsNumeric(MyContents) And Len(MyContents) = NumLgth Then
      MyContents = Format$(MyContents,"###-##-####")
      Source.DeleteContents
      Source.InsertText MyContents

      ' If the entry is not 11 or 9 characters, display a
      ' message box with the value of the String constant
      ' MsgTxt to ask the user to enter a valid social
      ' security number.
  Else
      MessageBox MsgTxt
      Source.Goto(True)

  End If

End Sub
```

Creating a custom menu item

You may want to create a script that accomplishes a repetitive task. If you make the task an item on a custom menu, you can decrease the number of steps that a user must complete in order to accomplish the task.

The following example creates a custom menu named My Menu with three menu items: First Menu Option, Second Menu Option, and Third Menu Option. It details how to associate a script with a menu item by calling the MenuSelection sub if the user chooses one of the menu items. The example also illustrates how to create a separator line in a custom menu.

If you want these custom menus to be available every time a user starts Word Pro, you need to make the custom menu script a startup script. For more information on startup scripts, search on “Scripts, startup” in the Word Pro Help Index.

Note The text of this script is stored in DW10_S7.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
Sub CustomMenu
   ' Set constants for the main menu and menu item titles.
   Const MenuName = "&My Menu"
   Const Option1 = "&First Menu Option"
   Const Option2 = "&Second Menu Option"
   Const Option3 = "&Third Menu Option"

   ' Create two MenuItem variables.
   Dim CrntMenu As MenuItem
   Dim MyMenu As MenuItem

   ' Set the variable CrntMenu to the MenuItem object
   ' contained in the Word Pro main menu property
   ' LwpMenuBar.
   Set CrntMenu = .ApplicationWindow.LwpMenuBar

   ' Check to see if the menu item already exists and delete
   ' it to prevent duplicates.
   CrntMenu.DeleteItem MenuName

   ' Create MyMenu as a main menu item on the Word Pro menu
   ' bar. Set the 3rd parameter of NewItem to False. Specify
   ' where MyMenu will display by giving the 4th parameter
   ' the name of the main menu item that should follow it.
   CrntMenu.NewItem MenuName,"",False,"&Help"

   ' Set the variable MyMenu to newly created main menu item.
   Set MyMenu = CrntMenu.Items.Item(MenuName)
   ' Add the first two menu items to the new main menu item
   ' and associate the menu items with the sub MenuSelection.
   MyMenu.NewItem Option1,"!MenuSelection",,
   MyMenu.NewItem Option2,"!MenuSelection",,

   ' Create a separator line on the menu.
   MyMenu.NewItem "-","",,

   ' Add the last menu item to the new main menu item and
   ' associate the menu items with the sub MenuSelection.
   MyMenu.NewItem Option3,"!MenuSelection",,
End Sub

Sub MenuSelection

' Create a message box to be displayed when a menu
' item is selected.
   MessageBox "A Menu Item was Selected!",64,"Menu Example"

End Sub
```

Setting custom views

Users often have preferences in how they like to view documents. This example creates a custom view using the Word Pro WinViewPrefs object.

The script first creates a split view and then displays the same document in two windows. In the top window, it displays the document at page width. In the bottom window, it displays the document in a multiple-page view showing the first seven pages.

This script can run when a menu option is chosen or when the user clicks a custom icon. For more information on running a script, see “Running The Memo Signing Script” in Chapter 4.

Note The text of this script is stored in DW10_S8.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
Option Public
%Include "WPBITMSK.LSS"

Sub SetViews

  ' Clear any current document splits.
  CurrentWindow.WinViewPrefs.ClearSplits

  ' Make the current window display 7 pages left to
  ' right by creating 7 columns in the window, turning off
  ' draft mode, and setting the view type to multiple pages.
  CurrentWindow.WinViewPrefs.NumCols = 7
  CurrentWindow.WinViewPrefs.IsInDraft = False
  CurrentWindow.WinViewPrefs.ViewType = LwpViewsMuliplePages
  ' Create a new window that uses 66% of the
  ' document window.
  .ApplicationWindow.UserInterfacePrefs.VerticalSplitWindow _
     = True
  .ApplicationWindow.UserInterfacePrefs.SplitPercentage = 66

  ' Display the new window.
  .NewWindow

  ' Makes the new window display with the page width view by
  ' turning off vertical splitting and draft mode,
  ' and by setting the view type to page width.
  .ApplicationWindow.UserInterfacePrefs.VerticalSplitWindow _
     = False
  CurrentWindow.WinViewPrefs.IsInDraft = False
  CurrentWindow.WinViewPrefs.ViewType = LwpViewsPageWidth
End Sub
```

Changing the function key setup

In Ami Pro, function keys are associated with the paragraph styles available in the current style sheet. In Word Pro, function keys are associated with the CycleKey setup. (For more information on CycleKeys, search on “CycleKeys” in Word Pro Help Index.) Using LotusScript, you can once again associate the function keys with styles in the current style sheet (SmartMaster).

The following example runs every time a user presses a key. If the key is a function key, the script suppresses the Word Pro CycleKey feature and changes the style of the current paragraph to the style indicated in the script.

If you want styles to be associated with function keys every time a user starts Word Pro, you need to make this example a startup script. For more information on startup scripts, search on “Scripts, startup” in the Word Pro Help Index.

Note The text of this script is stored in DW10_S9.LSS in the sample files directory. To save time, import the file into the Main sub by choosing File - Import Script in the IDE.

```
' These constants are in the (Declarations) script of the
' !Globals object. They are the numeric equivalents of the
' function key values.

Const F2  = 113
Const F3  = 114
Const F4  = 115
```

```
Sub Main
   ' Every time the user presses a key, run the
   ' sub MyKeyStroke.
   On Event KeyStroke From CurrentApplication Call _
      MyKeyStroke
End Sub

Sub MyKeyStroke(Source As WpApplication, Key As Integer, _
   Modifier As Integer, ReceivingLayout As String)

   ' If the user presses the F2, F3, or F4 key
   Select Case Key
      Case F2
         ' Change the style to Default Text.
         .Text.ParagraphStyleName = "Default Text"
         ' Prevents Word Pro from processing keystrokes
         ' as CycleKeys.
         End 2
      Case F3
         ' Change the style to Body Single.
         .Text.ParagraphStyleName = "Body Single"
         End 2
      Case F4
         ' Change the style to Bullet 1.
         .Text.ParagraphStyleName = "Bullet 1"
         End 2
   End Select
End Sub
```

Automating a merge

You can use LotusScript and Word Pro to merge variable data in one file (such as names and addresses) with text in another file. The data file can be a Word Pro data file (.LWP) or it can be an Approach data file (.APR).

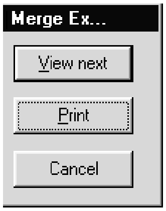
In the following example, the script uses a custom dialog box and an existing data file DW10_D10.LWP (located in the sample files directory) to illustrate how to do the following:

- Insert merge fields in a Word Pro document
- View merge data
- Print merged data in a document

To run this example:

1. Copy DW10_D10.LWP into the default document path specified in the Word Pro Preferences dialog box on the Locations panel.
2. In Word Pro, open the example document, DW10_S10.LWP.

Note DW10_S10.LWP is available in the sample files directory, and it contains a dialog box that works with this example. You can use this dialog box, or you can create your own.

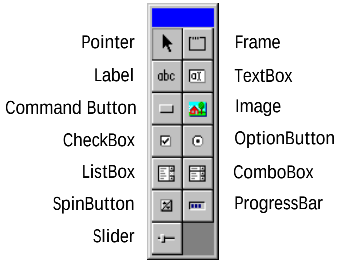


3. Choose Edit - Scripts & Macro - Run.
4. Select “Run script saved in the current file.”
5. Click OK.

```
Option Public
%Include "WPBITMSK.LSS"

Sub Main
   ' Declare a variable to hold the merge data file
   ' name and a variable to hold the status of opening
   ' the merge data file.
   Dim DataFile As String
   Dim Status As Integer

   ' Set the variable DataFile to the default document path.
   DataFile = .ApplicationWindow.UserInterfacePrefs.DocPath

   ' Add the name of the merge data file to the
   ' default document path.
   DataFile = DataFile & "\DW10_D10.LWP"

   ' Initialize the merge process.
   ApplicationWindow.ActiveDocument.MergeOptions.MergeStepNumber = _
      $LwpMergeStepNumber1

   ' Link the merge data file to the merge document. If an
   ' error occurs, exit the sub.
   Status = .MergeSetDataFile (DataFile, 0)
      If Status = False Then
         Exit Sub
      End If

   ' Insert the merge fields into the current document.
   ' Format the fields appropriately.
   .InsertField "TITLE"
   .Text.InsertHardSpace
   .InsertField "FIRSTNAME"
   .Text.InsertHardSpace
   .InsertField "LASTNAME"
   .Text.SplitParagraph
   .InsertField "ADDRESS"
   .Text.SplitParagraph
   .InsertField "CSZ"
   .Text.SplitParagraph
   .InsertField "FAVORITESPORT"

   ' Start the merge process.
   .ApplicationWindow.ActiveDocument.MergeOptions.MergeStepNumber = _
      $LwpMergeStepNumber3

   ' Show the custom dialog box.
   MergeDialog.Show
End Sub

' This sub prints the current record and displays the next
' record in the file.
Sub MergePrint
   ' Print the current record.
   .Print

   ' Merge the next record.
   .Merge $LwpMergeActionNextRecord
   .Merge $LwpMergeActionMergeOne
End Sub

' Merges the current record and displays the next record in
' the file.
Sub MergeViewNext
   ' Set merge options to view next.
   .ApplicationWindow.ActiveDocument.MergeOptions.Options = _
      LwpMergeOptFlgMergeViewAndPrint

   ' Merge the next record.
   .Merge $LwpMergeActionNextRecord
   .Merge $LwpMergeActionMergeOne

End Sub

' This sub closes the custom dialog box. It is attached
' to the Cancel button.
Sub Click(Source As LotusCommandButton)
   .ApplicationWindow.ActiveDocument.MergeOptions.Options = _
      LwpMergeOptFlgMergeViewAndPrint
   .ApplicationWindow.ActiveDocument.MergeOptions.MergeStepNumber = _
      $LwpMergeStepNumber2
   .Merge $LwpMergeActionClose
   Source.Parent.Close
End Sub
```

```
' This sub calls the MergeViewNext sub when the View next
' button is clicked.
Sub Click(Source As LotusCommandButton)
   Call MergeViewNext
End Sub

' This sub calls the MergePrint sub when the Print
' button is clicked.
Sub Click(Source As LotusCommandButton)
   Call MergePrint
End Sub
```

Index

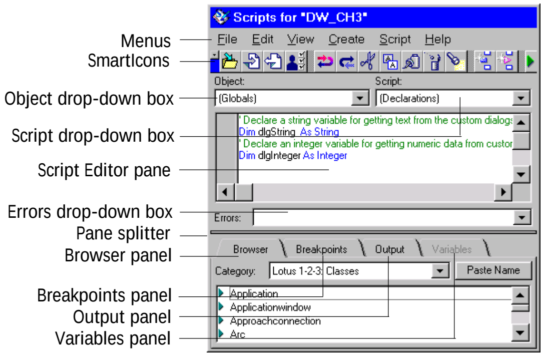


Lotus Controls Toolbox

The Lotus Controls Toolbox displays the icons for dialog box controls, such as text boxes, check boxes, option buttons, and so on. Click an icon to add the control to a dialog box.

The LotusScript IDE

The LotusScript Integrated Development Environment (IDE) provides a powerful and easy-to-use set of tools for creating and debugging scripts in Lotus products.



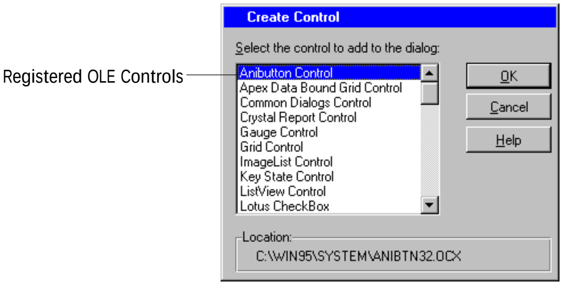
Object: (Globals)

Script: (Declarations)

```
' Declare a string variable for getting text from the custom dialogs
Dim dlgString  As String
' Declare an integer variable for getting numeric data from custor
Dim dlgInteger As Integer
```

Category: Lotus 1-2-3: Classes

- Application
- Applicationwindow
- Approachconnection
- Arc



Select the control to add to the dialog:

- Anibutton Control
- Apex Data Bound Grid Control
- Common Dialogs Control
- Crystal Report Control
- Gauge Control
- Grid Control
- ImageList Control
- Key State Control
- ListView Control
- Lotus CheckBox

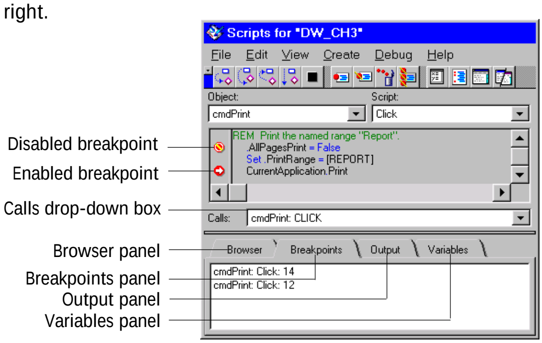
Location:
C:\WIN95\SYSTEM\ANIBTN32.OCX

Registered OLE Controls (OCX's)

You can add to a dialog box any third-party OLE Control (OCX) installed on your system. Third-party controls provide specialized features or enhanced versions of the standard controls. To add a third-party control, choose Create - Control - More and select a control from the list of available controls.

The Script Debugger

When an enabled breakpoint is reached during debugging, script execution is stopped and the script containing the breakpoint is displayed in the Script Debugger. You can display another procedure by selecting it in the Calls drop-down box at the bottom of the Debugger pane or in the Script drop-down box at the top right.



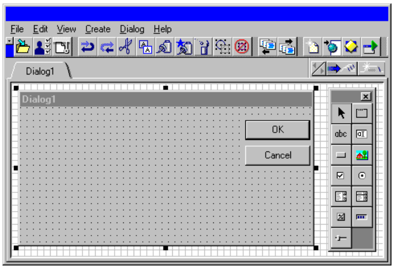
Object: cmdPrint

Script: Click

```
REM  Print the named range "Report".
   .AllPagesPrint = False
   Set .PrintRange = [REPORT]
   CurrentApplication.Print
```

Calls: cmdPrint: CLICK

cmdPrint: Click: 14
cmdPrint: Click: 12



LotusScript Dialog Editor

The LotusScript Dialog Editor provides a powerful and easy-to-use set of tools for creating and writing scripts for dialog boxes used in Lotus products.

Lotus®
Working Together®

Part No. 12773